The field of software, has, to date, focused almost exclusively on application-independent approaches. In this research, we demonstrate the role of application domain knowledge in the processes used to comprehend computer programs. Our research sought to reconcile two apparently conflicting theories of computer program comprehension by proposing a key role for knowledge of the application domain under examination. We argue that programmers use more top-down comprehension processes when they are familiar with the application domain. When the application domain is unfamiliar, programmers use processes that are more bottom-up in nature. We conducted a protocol analysis study of 24 professional programmers comprehending programs in familiar and unfamiliar application domains. Our findings confirm our thesis.

1. Introduction

"... we may have made nearly as much progress as we can reasonably expect in domain-independent languages, methods and tools. Nobody talks about domain-independence in hardware engineering because its disciplines differ so greatly in their underlying principles and skills. The same is true of many application domains that are well suited to software implementations" (Potts 1993).

Throughout the history of computing, attention has been devoted almost entirely to the development of application-independent approaches to software. Since it is clear that every computer-based solution is developed in response to a real-world problem, it is perhaps inevitable that, in addition to the solution, we must eventually turn our attention to the problem, or application domain.

In this paper, we address the issue of application-dependence in the context of comprehending computer programs. Like the majority of researchers before us, by
comprehension we mean, simply, understanding existing program code (see, for example, Pennington (1987), Wiedenbeck (1991)). Comprehending computer programs is a task that underlies many programming tasks. Comprehension plays a major role in the testing and debugging phases of program development, and in later program maintenance and enhancement phases. To enhance a program, for example, a programmer must understand what the program does and how it does it, prior to the necessary testing and debugging, which also involve comprehension.

Despite the fact that certain authors have noted differences in computer program comprehension based on specific computer programs (see, for example, Curtis et al. 1989), no research to date has explicitly investigated the effect of application domain knowledge on program comprehension per se. It is our contention that familiarity with the application domain plays a major role in the way a programmer seeks to understand a computer program, i.e., the comprehension process. Knowledge of the application domain permits programmers to demonstrate expertise by recognizing meaningful patterns (or chunks of information) (Chi et al. 1982). When programmers are familiar with the application domain, therefore, they are able to determine (i.e., hypothesize) the function of the program, and of modules within the program without examining program statements in detail. In this situation, they are able to use comprehension processes that are more top-down in nature. When programmers are unfamiliar with the program, they must examine it at the level of detailed code; they infer the purpose of program modules and the program as a whole on the basis of their understanding of the program details. They use comprehension processes that are more bottom-up in nature and therefore require more effort overall.

We tested our thesis with professional programmers in the accounting domain by examining the program comprehension processes used in understanding programs in the familiar accounting domain and the unfamiliar hydrology domain.

2. Prior Research into the Computer Program Comprehension Process

Although program comprehension plays a fundamental role in many programming tasks, relatively few studies have addressed the comprehension process. Here, we present the prevailing theories of the comprehension process, together with the empirical support for those theories. (For a thorough review of computer program comprehension research see Boehm-Davis (1988).)

2.1. Current Theories of the Computer Program Comprehension Process

Brooks (1983) views comprehension as a top-down process in which programmers use their existing programming knowledge to develop increasingly specific hypotheses about the function of the program. Programmers first develop a global hypothesis about the overall program function based on as little information as the title. It is impossible to verify such high-level hypotheses directly. Programmers, therefore, develop successively more specific, subsidiary hypotheses in a hierarchical fashion until a level of specificity is reached. This level of specificity permits programmers to develop hypotheses about certain structures and operations that can be expected to occur in the program or related documentation. Programmers then seek to verify these low-level hypotheses based on relevant information in the program. Hypotheses are developed, modified, or rejected (and reevaluated) until an adequate understanding of the program has been developed.

The verification process links the hypotheses to the program text and is therefore a critical part of the theory. Programmers verify hypotheses by searching for what
Brooks calls "beacons," i.e., "sets of features that typically indicate the occurrence of certain structures or operations in the code" (p. 548). Beacons are related to programming plans (Soloway and Ehrlich 1984); stereotyped patterns of program instructions for achieving a single, well-defined function. A number of these simple sets of program instructions are combined into a more complex plan, i.e., a program. When a hypothesis is confirmed, the specific features in the code are bound to the functional specifications; when a hypothesis is not confirmed, it is reevaluated and the process is repeated. The process is complete when all parts of the program are bound to the subsidiary hypotheses at the base of the hypothesis hierarchy.

Brooks' theory of comprehension has been subjected to substantial empirical scrutiny. The role of beacons in the hypothesis verification process has been the focus of attention in studies by Wiedenbeck (1986, 1991) and Gellenbeck and Cook (1991). All of these studies (a total of eight) support the notion of beacons.

The second theory of computer program comprehension views comprehension as a bottom-up process. Early bottom-up models (e.g., Shneiderman and Mayer 1979) proposed that understanding programs is a process of program reconstruction based on a bottom-up, chunking process, i.e., programmers first understand program statements, and then gradually build up to an understanding of groups of statements, or modules, and so on. More recently, Pennington (1987) has approached the program comprehension process by inferring process from the nature of the programmer's mental representation of different types of information inherent in a program (i.e., control flow, data flow, function, and state). Pennington considers two types of programming knowledge as the basis for programmers' mental representations: knowledge of the structure of the text and knowledge of programming plans. Text structure knowledge relates to the representation of the program in terms of a limited number of control constructs (usually sequence, selection, and iteration). Programming plan knowledge is based on the fact that programmers associate certain patterns of program instructions with the accomplishment of certain functions. Note that Brooks' theory is based on the latter alone.

Pennington conducted a study to investigate the nature of programmers' mental representations. Specifically, she examined which types of program information were represented most effectively in programmers' memories following a period of study. From this knowledge of program information, she then inferred the nature of the mental representation and the processes used to develop that representation. Her results support text structure knowledge as the primary organizing characteristic, leading her to conclude that the "meaning of program text is developed largely from the bottom-up" (1987, p. 326).

2.2. Evaluation of Current Theories of the Program Comprehension Process

There is, then, empirical evidence to support two opposing theories of computer program comprehension. It seems likely that a further variable, unidentified in the current theories, determines the nature of the comprehension process a programmer uses. It is the thesis of this research that the missing, or uncontrolled, variable is the programmer's knowledge of the application domain.

Clearly, the use of top-down problem-solving processes is a more parsimonious approach than the use of bottom-up processes. Hence, we expect that programmers will use a more top-down comprehension process when possible in order to conserve effort. Further, we expect that more experienced programmers will have more
effective processes at their disposal and therefore will be able to conserve effort better than less experienced programmers. We argue that the type of experience that is relevant in these circumstances is the programmer’s knowledge of the domain of the application under consideration. When programmers are unfamiliar with the application domain, they will not be able to draw upon relevant templates or schema from their knowledge bases. Therefore, they will not be able to construct a global hypothesis relating to program function. Instead, they will need to engage in an in-depth examination of the program to determine from the bottom-up the function that the program performs.

We can test this assertion ex post by examining whether the nature of the application domain played a role in the empirical studies conducted to assess support for the extant theories. First, as noted in §2.1, all studies conducted to assess support for Brooks’ top-down theory investigated factors involving the detection of beacons. Wiedenbeck (1986, 1991) conducted a total of six studies: four of the six studies used a shellsort; one used a binary search; the final experiment used an odd-even transposition sort and a distribution sort. Similarly, Gellenbeck and Cook (1991) used binary search and sort procedures.

Note that the programs used by both Wiedenbeck and Gellenbeck and Cook were approximately 20 lines in length and therefore contained little functionality apart from sorting and searching. Further, sort and search routines are used in many different programs in many different application domains. The structure of such programs will, therefore, be application domain-independent, i.e., a sort program is a sort program irrespective of application domain. In such circumstances, knowledge relevant to beacons resides solely in the programming domain, not in the application domain. Since programmers possess all the knowledge they need to understand these programs, our thesis suggests that programmers would use more top-down comprehension processes, and this was, in fact, the case.

Second, Pennington’s (1987) test of her bottom-up theory used a program that tracked and computed specifications for industrial plant designs and is rich in specialized knowledge about industrial design. Further, Pennington made no attempt to match, or otherwise control for, the application domain of the program and the programmers’ knowledge of that domain. It is unlikely, therefore, that programmers in this experiment had prior knowledge of the specific application domain under investigation. Our thesis suggests that programmers would use more bottom-up comprehension processes under these circumstances and this was the case.

Hence there is ex post support for our thesis based on empirical studies reported in the literature. What is needed now is an ex ante investigation of our theory.

3. The Current Study

Figure 1 presents our model of the comprehension process. Its foundation is Letovsky’s (1986) view of programmers as “knowledge-based understanders.” It consists of essentially three components: (1) a knowledge base, which is comprised of both programming knowledge and application domain knowledge; (2) a mental model, which is a programmer’s internal or mental representation of the computer program and the way in which it functions, i.e., it represents both static and dynamic aspects of the programmer’s knowledge; and (3) the assimilation or comprehension process, which Letovsky (nonspecifically) states can be top-down or bottom-up.

Our thesis is that the level of application domain knowledge determines whether the
A programmer can use a more top-down comprehension process. When the application domain of the program is familiar to programmers, they are able to generate a primary hypothesis about the nature of the program and comprehension will most likely proceed in the more top-down, deductive manner described by Brooks (1983). However, when the application domain is unfamiliar, programmers do not possess the relevant application domain knowledge to engage in top-down comprehension. Instead, programmers will most likely use a more bottom-up process to develop their understanding of the program inductively, examining program statements line-by-line.

Although research to date has conceived of the comprehension process as predominantly top-down or bottom-up in nature, it is unlikely that a programmer faced with the task of understanding a computer program will use one approach to the exclusion of the other. Since a more top-down comprehension process is hypothesis-driven, a programmer using this approach is expected to state hypotheses about the function of the program and to scan the program to verify the hypotheses (see Brooks 1983). However, in the process of verifying an hypothesis, the programmer may notice features of the program that are unrelated to the hypothesis and therefore may be motivated to make inferences about the program.

Since a more bottom-up comprehension process is based on making inferences from the program itself, programmers using this approach are expected to state inferences about the program as they build up their knowledge. However, in the process of understanding, programmers will develop their knowledge and may become sufficiently confident in their knowledge of the program to state an hypothesis. Therefore, the comprehension process is viewed as continuous rather than absolute. We refer to comprehension processes that are more top-down or more bottom-up in nature and we state the following proposition.

**Proposition.** A programmer seeking to understand a computer program in a familiar application domain uses a more top-down comprehension process than for a program in an unfamiliar application domain.
4. Methodology

We conducted a process tracing study in which programmers studied computer programs from two application domains: one familiar, the other unfamiliar. Concurrent verbal protocol analysis was used to examine the comprehension process since it provides a rich set of data about cognitive processes in a relatively natural working environment. To minimize potential validity problems, we trained subjects to provide verbal data, instructed them only to "think-aloud" during problem solving and not to explain their thought processes, and did not probe for specific facts (Russo et al. 1989).

4.1. Task Setting

Accounting was selected as the familiar application domain because of the ubiquitous nature of accounting applications and therefore the ease of identifying potential subjects. Hydrology was selected as the unfamiliar application domain because it is unrelated to accounting, and therefore it was easy to identify subjects with accounting experience who did not have hydrology experience.

COBOL was the programming language investigated because of its predominance as a business programming language. An estimated $77 \times 10^9$ lines of COBOL code are currently in use in the U.S. (Bennett et al. 1988). Since such COBOL programs represent a significant corporate asset, the continued importance of knowledge concerning the comprehension of COBOL programs is assured for the foreseeable future.

4.2. Subjects

Twenty-four information systems professionals employed in developing and/or maintaining COBOL accounting applications participated in the study. The average participant was 36.5 years of age, had 10.7 years experience as an IS professional, knew 5 programming languages, and was male in 4 out of 5 instances. The average self-reported score for COBOL knowledge was 4.7 on a 5-point scale. None of the subjects was familiar with the hydrology domain as determined by a background questionnaire.

4.3. Experimental Materials

The experiment required a program to familiarize participants with the experimental procedures and a computer program in each of the application domains. The programs chosen accessed all files sequentially and contained internal sorts.

The practice program was a simple sort program of 99 lines of code (33 lines in the PROCEDURE DIVISION) that sorted an input inventory file and prepared a report that listed inventory levels for all items in a prescribed order.

The accounting program was a modified version of an operational payroll program, which computed and produced paychecks, mailing labels, and a payroster. The sort sequenced timecards by location, last name, and social security number. Modifications were made to facilitate comprehension: (1) by modifying certain paragraph and variable names and reordering paragraphs; and (2) to ensure comparable complexity in the two experimental programs (see below).

The hydrology program was a modified version of an operational water quality management program, which computed averages and variances for seven parameters (coliform, nitrate, chloride, lead, fluoride, pH, and alkalinity) based on test results. The sort organized the test information by well identification number, year, and month. Similar modifications were carried out to improve comprehension. Further
changes were also made since the program had become difficult to understand due to changes made over time.

The programs needed to be of equivalent complexity so that any performance differences could be attributed to application domain knowledge. Since many different program complexity measures are highly correlated with source lines of code (SLOC), we used SLOC as our complexity standard (Lind and Vairavan 1989). The total SLOC for the accounting and hydrology programs were 417 and 416, respectively. Breakdowns based on DIVISIONS (IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE) were as follows: 3, 20, 234, 160 (accounting program) and 3, 18, 233, 162 (hydrology program). Hence, the complexity of the two programs was similar based on SLOC.

4.4. Experimental Design

A within-subjects design, with application domain as the within-subjects factor, was used to investigate the proposition. A further check on application domain familiarity was made by asking subjects to rate their familiarity with the application domain after they had studied each of the programs. A paired \( t \)-test of differences in responses showed that subjects were indeed more familiar with the accounting domain \( (t = 5.84; p < 0.0001; df = 23) \).

Hypotheses and inferences were used to distinguish between top-down and bottom-up comprehension processes (Brooks 1983). A more top-down process is hypothesis-driven and is therefore characterized by relatively more hypotheses than inferences. A hypothesis occurs when a programmer states an expectation about the purpose of the program based on limited information. Therefore, a hypothesis is usually stated in the form of a conjecture. In the following protocol segment, for example:

35. So now we're printing another report here
36. Probably a payroll register.

the second statement is a hypothesis because the programmer, based on limited examination of the program, conjectures that the report is a payroll register.

A more bottom-up process is inferential in nature and is therefore characterized by relatively more inferences than hypotheses. An inference occurs when a programmer builds up to some understanding based on examination of the computer program. Thus, an inference is usually stated in the form of a conclusion. In the following protocol segment, for example:

13. SORT file is going to be 80
14. Oh, some type of well on that
15. Must be doing water.

the last statement is an inference. In the two prior statements the programmer noticed that well information is defined in the sort file. From that specific knowledge, the programmer concluded (correctly) that the program is a water (hydrology) application.

The nature of the comprehension process was determined by the process score, obtained by subtracting the number of inferences from the number of hypotheses; therefore, the more positive the process score, the more top-down the process. Process scores were computed for each comprehension session (two for each programmer, one in each domain). To assess support for the proposition, we tested the following specific hypothesis.
HYPOTHESIS. *A programmer's process score is higher in the accounting (familiar) than in the hydrology (unfamiliar) application domain.*

4.5. Experimental Procedure

We conducted a pilot study prior to the main study to test the experimental materials and procedure, and to determine the length of the comprehension period. Based on the pilot study, programmers in the main study were permitted 15 minutes for comprehension in each application domain. Since we wished to analyze the type of knowledge participants heeded, they were simply instructed to “understand” the program.

In the main study, subjects first completed a mailed background questionnaire designed to determine their suitability as potential subjects. Since concurrent verbal protocols were the major source of data in this study, subjects were run individually through the experiment. The experiment then took place in three segments, a practice session (to familiarize subjects with the process of thinking aloud and the experimental procedures) and two experimental sessions (one in each application domain). The order of presentation of the application domains was counterbalanced such that half of the subjects worked first on each of the two programs.

4.6. Verbal Protocol Data

The verbal protocol data was analyzed to determine the nature of the comprehension process for the programs in the familiar and unfamiliar application domains. First, we prepared a coder's manual that contained detailed definitions, examples of hypotheses and inferences, and rules for assigning codes. The appendix provides a summary of the instructions in the coder's manual. Next, we trained two coders who were familiar with COBOL, but unfamiliar with the research hypotheses to code protocols using the pilot study data. Both coders then coded all 48 protocols from the study proper (2 from each of 24 participants). The interrater reliability for both pilot and study coding was assessed using Cohen's Kappa (1960) and judged according to levels of acceptability suggested by Landis and Koch (1977). Coders were trained to "substantial" levels of agreement (Kappa ≥ 0.61). The level of agreement for the study protocols was highly significant (z = 34.37, p < 0.001) and is considered "substantial" at 0.66 (raw agreement was 0.96). Disagreements between coders were reconciled in joint meetings among the two coders and the first author.

5. Data Analysis

We used the coded verbal protocol data to assess support for our hypothesis. Table 1 presents the process scores for each programmer in each application domain as well as the difference in the scores. The more positive the difference score, the more top-down the process used in the familiar domain compared with the unfamiliar application domain.

A one-tailed, paired t-test was used to test the hypothesis that programmers use a more top-down comprehension process in a familiar application domain. The test was significant \( t = 2.10; p < 0.024; df = 23 \). Our Hypothesis, and therefore our Proposition, is supported.

We illustrate the differences in comprehension processes used in the familiar and unfamiliar application domains with protocol excerpts. The first excerpt is from the protocol of subject 9 working in the familiar application domain. His use of a more top-down process is illustrated by the hypotheses he stated in assertions 10, 38, and...
TABLE 1
Hypotheses and Inferences for Each Programmer in Each Domain

<table>
<thead>
<tr>
<th>Subject ID</th>
<th>Familiar Domain</th>
<th>Unfamiliar Domain</th>
<th>Difference Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Hypotheses</td>
<td># Inferences</td>
<td># Hypotheses</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>(2)(^\d)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>(7)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>(7)</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>(5)</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>(1)</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>(2)</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>(2)</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>(1)</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>(1)</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>(7)</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>(-2)</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>(-3)</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>(2)</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>(2)</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>(2)</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>(2)</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>(3)</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>(6)</td>
<td>7</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>(2)</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>(1)</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>13</td>
<td>(5)</td>
<td>8</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>(-1)</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>(3)</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^1\) Process Score
In assertion 10, while familiarizing himself with file names, the programmer correctly hypothesized the program’s overall purpose—payroll. Assertions 38 and 40 produced further refinements of the purpose of the program—university payroll, student payroll. Note, that this protocol provides explicit support for Brook’s (1983) notion of a hierarchy of hypotheses.

1. O-K, I’m looking at the program, I can’t tell anything by the name, EXAF
2. Looking down it has a MASTERFILE
3. It looks like all the files are sequential
4. MASTERFILE
5. TIMECARDS
6. ROSTERFILE
7. ORGANIZATION
8. LABELSFILE
9. CHECKFILE
10. It looks like a payroll or some type of thing like that

37. DEPT-GROSS, UNIVERSITY-GROSS
38. Ah, must be a university payroll, maybe
39. MASTER-RECORD-IN
40. That must be what the STUD is. It’s probably student payroll.

The second excerpt, from the same programmer studying the program in the unfamiliar application domain, illustrates the programmer making inferences about the program. In assertion 145, for example, he inferred that the purpose of a working-storage variable is the prime, or initial value, used to make comparisons. These excerpts from the same programmer in different domains clearly illustrate the nature of hypotheses and inferences and therefore of top-down and bottom-up comprehension processes.

141. There’s a MATCH-WELL-ID
142. If matched, if matched it checks to see if it matched again, of course it’s after it does FINISH-MONTH
143. Oh, it read another well
144. So that test
145. So WORKING-STORAGE must be the prime
146. working site where was it initialized?

The difficulty many programmers had in formulating hypotheses about the unfamiliar hydrology program is illustrated in the protocol of subject 24 presented in the final excerpt. This segment demonstrates that the programmer was unable to ascertain the purpose of the program and therefore unable to formulate an hypothesis. In assertion 56, the programmer, who worked for an oil company, commented that someone in production (i.e., oil production) would be better able to “relate to” this program, clearly indicating that he believed application domain knowledge is important to program comprehension.

56. Hum, OK, looks more like a program somebody in production could relate to
57. Except they would have written it in PL/1
58. It looks like the program is trying to create a report or series of reports
6. Discussion

This study addressed a general notion of current topical interest in the area of systems development, viz., the importance of application domain knowledge to the systems development process. We investigated this notion in the context of the processes programmers with varying application domain knowledge use to comprehend computer programs.

Two current theories of computer program comprehension propose the use of top-down or bottom-up comprehension processes, but neither attempts to address the driving force behind these apparently diverse approaches. We hypothesized that programmers who are knowledgeable regarding the application domain are able to use the more parsimonious top-down comprehension processes. When they do not possess that knowledge, programmers use more bottom-up processes that involve examining the code in detail. We tested this thesis by examining the extent to which programmers stated hypotheses (indicative of a top-down process) and made inferences (indicative of a bottom-up process) when seeking to understand programs in familiar and unfamiliar domains. We assessed our thesis using professional programmers familiar with the accounting domain and unfamiliar with the hydrology domain. The thesis was supported. Further evidence could be provided by examining professional programmers familiar with hydrology and unfamiliar with accounting, for example.

The potential limitations of the study center on the sample size and the size of the computer programs used as task materials. First, a small number of participants is the norm for protocol analysis studies. Note, however, that the sample size is larger than in the majority of protocol analysis studies reported in the literature. To offset the small sample size, we used a within-subjects design. Second, the programs used as task materials were fairly small compared with the size of systems that exist in practice. However, these programs were at least 1.5 times as large as those used in previous studies of program comprehension (Pennington 1987) and 20 times as large as those of Wiedenbeck (1986, 1991). Hence, this limitation is also a strength when compared with similar studies.

The strengths of this study are the use of professional programmers and the formal coding of the protocol data. Prior to this study, Pennington’s (1987) study is, perhaps, the only published study of program comprehension, per se, to use practicing programmers. (Note that studies of programmers that involved, but did not directly address comprehension, have used professional programmers.) Also, most protocol analysis studies in this area examined only a fraction of the protocols in detail, and none of them formally coded their data.

The findings have implications for the conduct of research on program comprehension, in particular, and on systems development, in general. Our findings demonstrate the need to consider not only the programming but also the application domain. The majority of “programming” studies to date have used simple programs where knowledge of the application domain has not been an issue. The use of such programs is justified in theoretically-based studies of the programming domain and in studies of the programming domain in which application domain knowledge is controlled.

This study also demonstrates the importance of application domain knowledge in software development, in general. Recent literature has begun to stress the importance of the application domain (Glass and Vessey 1992). However, there are few systematic studies of the role that application domain knowledge plays in software
development. Given the importance of application domain knowledge in computer programming (a software development task that is considered to be relatively remote from the original application domain), we believe that application domain knowledge may play an even larger role in other software development tasks. In fact, Vessey and Conger (1993) also report that knowledge of the application improved novice systems analysts' ability to specify information requirements. Our study highlights the importance of carefully evaluating the appropriateness of using context-free methods, tools, and techniques in all systems development tasks.*

Acknowledgements. The authors wish to thank Rich Carlson, Duncan Fong, Robert L. Glass, Ed Ketz, Ben Kleindorfer, and Mark Sharfman for their comments on earlier versions of this work and Mitchell Todd and Heidi Schultz for research assistance. The Center for Interdisciplinary Research in Information Systems in the College of Business Administration at The Pennsylvania State University and the College of Business at The University of Tulsa provided financial assistance.

* Joyce Elam, Associate Editor. This paper was received on March 15, 1994, and has been with the authors 1 month for 1 revision.

Appendix—Summary of Instructions in Coder's Manual

The following is a brief summary of the instructions provided to the coders. The complete coder's manual contains general rules for assigning codes as well as instruction for a variety of difficult situations. The manual illustrates each situation with an example excerpted from the pilot protocols.

A hypothesis occurs when a programmer conjectures or states an expectation about the contents of the program based on limited information from the program. An hypothesis will generally be stated as a question or speculation. Hypotheses will often include such phrases and terms as "I guess," or "probably." A hypothesis is, in effect, a "hunch," or reasoned guess.

211. So we had a bunch of PARAMETERS, right, coming in,
212. which would be probably on missing WELLS,
213. Let me think here
214. And that was the key
215. No WELL-ID given, Yeah, that makes sense,

EXAMPLE A.1. In Example A.1, line 211 the programmer summarizes his or her understanding about a previous portion of the program and the phrase should not be coded (see below). Line 212 is coded as a hypothesis because the programmer makes the statement without knowing if it is true. Lines 213 through 215 contain phrases where the programmer considers the validity of the hypothesis stated in line 212. These phrases are not coded.

Hypotheses can be stated in terms of expected alternatives ("is there a yearly because we were on years or would that be for max?") and must address the program's functioning, not the physical layout. For example, the following statement would not be coded: "You would think that would be right in here." Finally, it is important that the programmer state an expectation about the program for a phrase to be coded as a hypothesis. A question, such as "and what does that do?" would not be coded.

An inference occurs when a programmer builds up to an understanding based on his or her examination of the computer program and is usually stated as a conclusion. A programmer typically pauses prior to stating an inference. The pause is usually indicated by terms such as "Oh," "Yeah," "So," "Looks," or "Okay." The use of such terms is not sufficient to code the phrase as an inference, but indicates that it should be considered.

117. Look up one of these,
118. GROSS-OUT
119. moves me out,
120. Yeah, it's just your stub and so on

* The complete coder's manual is available from the first author upon request.
Shaft • Vessey

EXAMPLE A.2. In line 117, the programmer states that he or she is about to look up a part of the program. In line 118 the programmer reads aloud. In line 119, several lines of code are summarized. Based on the portions of the program studied in lines 117 through 119, the programmer draws an inference in line 120 ("it's just your stub and so on"). Lines 117 through 119 are not coded. Line 120 is coded as an inference because the programmer states his or her understanding as a conclusion and the previous lines show that the programmer built up his or her knowledge by examining the program prior to making the inference. Simply rereading and rehashing particular lines of code does not indicate an inference.

For a phrase to be assigned a code indicating an inference, two things must be ruled out. First, the phrase cannot be just a translation from COBOL to English. For instance, referring to the LOAD-PARAMETERS paragraph as "load the parameters" does not indicate an inference. Second, phrases where a programmer summarizes his or her understanding of the program are not coded, such as when a programmer reads back through a portion of the program that he or she had already studied and draws no new conclusions. Consistent with coding hypotheses, to be assigned a code, the phrase must address the program's functioning, not the physical layout.

In situations where hypotheses and inferences are mixed in a single episode, the inference must be examined to determine if it was stated as part of verifying the hypothesis. If so, the phrase is coded as a hypothesis. If the inference was not stated as part of verification, the critical statement of the episode is examined closely and assigned a code depending on if the phrasing more closely matches that of an hypothesis or an inference. For instance:

198. Looking at the OUTPUT section again,
199. Seeing what gets out when,
200. Test Samples,
201. O.K. here some ERRORS,
202. I had no matching WELL RECORDS,
203. So if the PARAMETERS don't match, if there is an extra PARAMETER,
204. probably get sick, there wasn't a well coming in for that one. O.K. does that make sense?
205. Finish the Month

EXAMPLE A.3. In this example, line 204 is the critical statement. There appears to be some build up (lines 198–203) of knowledge prior to 204. In line 204 the programmer mixes the two kinds of statements, and then continues reading the program in line 205. The phrasing of the critical statement is examined to determine the correct code. Line 204 is phrased more closely to a hypothesis than an inference ("probably get sick") and a code of "H" should be assigned. Note, the programmer was unwilling to state what they knew in the form of a conclusion (inference), and instead phrased the statement as a conjecture (hypothesis).

References


The Case of Computer Program Comprehension


