Int. J. Human-Computer Studies (1999) **51,** 31–70
Article No. ijhc.1999.0268
Available online at http://www.idealibrary.com on IDEAL®

Program understanding behavior during corrective maintenance of large-scale software

A. Marie Vans
Hewlett-Packard Corporation, 3404 E. Harmony Road, Fort Collins, CO 80525, USA

Anneliese von Mayrhauser and Gabriel Somlo
Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873, USA

This paper reports on a software understanding field study of corrective maintenance of large-scale software. Participants were professional software maintenance engineers. The paper reports on the general understanding process, the types of actions programmers preferred during the debugging task, the level of abstraction at which they were working and the role of hypotheses in the debugging strategies they used. The results of the observation are also interpreted in terms of the information needs of these software engineers. We found that programmers work at all levels of abstraction (code, algorithm, application domain) about equally. They frequently switch between levels of abstraction. The programmers' main concerns are with what software does and how this is accomplished, not why software was built a certain way. These questions guide the work process. Information is sought and cross-referenced from a variety of sources from application domain concepts to code-related information, outpacing current maintenance environments' capabilities which are mostly stratified by information source, making cross-referencing difficult.

1. Introduction

Program understanding is an important aspect of a variety of maintenance tasks. During corrective maintenance, the programmer has to understand the software enough to analyse a problem, locate the bug and determine how it should best be fixed without breaking anything. For larger software products, understanding will, by necessity, be partial. Maintainers are not always experts in the application area. Even when they are, they may not be experts in the implementation language. In order to learn more about comprehension behavior during corrective maintenance, we observed professional maintenance programmers during corrective maintenance tasks on actual software products. The observations considered the two situations mentioned above. The questions we tried to answer are as follows.

1. What kinds of actions do programmers perform when debugging code?
2. Do programmers follow the integrated comprehension model of von Mayrhauser and Vans (1995*a*)? Do they switch between its three model components? Is there a preference for a particular model component?

1071-5819/99/070031 + 40 $30.00/0

3. Is it possible to identify a specific comprehension process that is common to the subjects and thus indicative of debugging tasks? What role do hypotheses play in the comprehension process?
4. Are there certain types of information programmers tend to look for during corrective maintenance?

We were particularly interested in programmer behavior in a realistic setting, doing actual corrective maintenance work on large-scale software. This interest is based on the premise that in industry, large-scale programs are the more prevalent focus for software comprehension and corrective maintenance activities. von Mayrhauser and Vans (1993*a*, *b*, 1995*a*) describe an integrated comprehension model that has been shown to model large-scale code comprehension activities as a combination of three comprehension processes at the program (code), situation (language independent, algorithmic) and domain (application) levels.

Section 2 provides background on the integrated comprehension model. Section 3 describes the experimental design. It is an observational field study of maintenance programmers in industry working on corrective maintenance. Software was at least 40 000 lines of code. Section 4 reports on the results of the observations with regard to the questions posed above. Section 5 summarizes conclusions and provides working hypotheses based on our results that should be evaluated with further experiments.

2. Comprehension model (von Mayrhauser & Vans, 1995*a*)

The current literature on program understanding assumes that comprehension proceeds either top-down, bottom-up or a combination of the two, and that a mental representation of the program is constructed as a result of these processes. Observations with large-scale code (von Mayrhauser & Vans, 1993*a*, *b*, 1995*a*) indicate that comprehension involves both top-down and bottom-up activities. Consequently, an integrated comprehension model (von Mayrhauser & Vans, 1995*a*) was developed that includes the following components: (1) *program model*, (2) *situation model*, (3) *domain model* (or *domain model*) and (4) *knowledge base*. Soloway and Ehrlich's model (Soloway, Adelson & Ehrlich, 1988) is the basis for the top–down component (the domain model) while Pennington's (1987*a*, *b*) model supports the program and situation models. The program, situation and domain models describe the processes and mental representations that lead to an understanding of software. Any of the three models may be accessed by any of the others. Beacons, goals, hypotheses and strategies determine the dynamics of the cognitive tasks and the switches between the models. Each model component includes the internal representation (mental model) of the program being understood. This representation differs in level of abstraction† for each model. Each model component also includes strategies to build this internal representation. The knowledge base furnishes each model with information related to the comprehension task. It also stores any new and inferred knowledge.

†By abstraction we mean the level of decomposition of the program code being understood.

The domain model represents knowledge schemas about an application domain. The *domain* model of program understanding is usually invoked if the application domain is familiar (Soloway & Ehrlich, 1984; Soloway *et al*., 1988). For example, a domain model of a distributed system would contain knowledge about client, server and middleware infrastructure (operating systems, graphical interfaces, network transport stacks, services) and how they interact with each other. This knowledge also includes specialized schemas such as design rationalization (e.g. the pros and cons of using the OMG CORBA standard vs. COM/DCOM for object communication across the network). A new distributed system will be easier to understand with such knowledge than without it. Domain knowledge provides a motherboard into which specific product knowledge can be integrated more easily. It can also lead to effective strategies to guide understanding (e.g. understanding server response to client requests requires understanding how the communication middleware interacts with the underlying network protocol stacks).

Hypotheses guide comprehension. Letovsky (1986) defines *Hypotheses* as conjectures and comprehension activities (actions) that take on the order of seconds or minutes to occur. *Actions* classify programmer activities, both implicit and explicit during a maintenance task. Examples of action types include "asking a question" and "generating a hypothesis". Letovsky identified three major types of hypotheses: *Why* conjectures hypothesize the purpose of some function or design choice. *How* conjectures hypothesize about the method for accomplishing a program goal. *What* conjectures hypothesize about what something is, for example a variable or function. Conjectures vary in their degree of certainty from uncertain guesses to almost certain conclusions. Brooks (1983) considers hypotheses the only drivers of cognition. Understanding is complete when the mental model consists of a complete hierarchy of hypotheses in which the lowest level hypotheses are either confirmed (against actual code or documentation) or fail. At the top is the *primary hypothesis*, a high-level description of the program function. Once the primary hypothesis exists, subsidiary hypotheses in support of the primary hypothesis are generated. The process continues until the mental model is built. Brooks considers three reasons why hypotheses fail: code to verify a hypothesis cannot be found, confusion due to a single piece of code that satisfies different hypotheses and code that cannot be explained. *Goals* or *Questions* (Letovsky, 1986) embody the cognitive processes by which maintenance engineers understood code. A goal can be explicit or inferred from a hypothesis. It involves formation of a hypothesis in support of reaching the goal (answering the question). Hypotheses then lead to supporting actions, lower-level goals, subsidiary hypotheses, etc. Hypotheses occur at all levels, the application domain, algorithmic and code levels.

When code to be understood is completely new to the programmer, Pennington (1987*a*, *b*) found that programmers first build a control flow abstraction of the program called the *program model*. Once the program model representation exists, Pennington showed that a *situation model* is developed. This representation, also built from the bottom up, uses the program model to create a data-flow/functional abstraction. The integrated model assumes that programmers can start building a mental model at any level that appears opportune. Further, programmers switch between any of the three model components during the comprehension process. When constructing the program model, a programmer may recognize clues (called *beacons*) in the code indicating a common task such as sorting. If, for example, a beacon leads to the hypothesis that

a sort is performed, the switch is to the domain model. The programmer then generates sub-goals to support the hypothesis and searches the code for clues to support these sub-goals. If the search finds a section of unrecognized code, the programmer jumps back to building the program model.

Comprehension is guided by systematic, opportunistic or a mixed systematic/opportunistic strategy. In a systematic approach, the programmer applies a systematic order to understanding code completely, for example, code comprehension line by line. An opportunistic approach studies code in an as-needed fashion. Littman, Pinto, Letovsky and Soloway (1986) found that programmers using a systematic approach to comprehension are more successful at modifying code (once they understand it). Unfortunately, the systematic strategy is unrealistic for large programs. A disadvantage to the opportunistic approach is that understanding is incomplete and code modifications based on this understanding can be error prone (Littman *et al*., 1986).

The above models have several commonalities. At the highest level of generality, all accommodate (1) a mental representation of the code, (2) a body of knowledge (knowledge base) stored in long-term memory and (3) a process for incorporating the knowledge in long-term memory with external information (such as code) into a mental representation. Each differs in the amount of detail for each of these three main components. Each of these models represent important aspects of code comprehension and many overlap in characteristics. For example, Brooks (1983) and Letovsky (1986) focus on hierarchical layers in the mental representations. Brooks (1983), Letovsky (1986) and Soloway and Ehrlich (1984, 1988) use a form of top-down program comprehension while Pennington (1987*a*, *b*) and Letovsky (1986) use a bottom-up approach to code understanding. All models use a matching process between what is already known (knowledge structures) and the artifact under study. No one model accounts for all the behavior we see when programmers understand unfamiliar code. However, we can take the best of these models and combine them into an *integrated comprehension model* that not only represents relevant portions of the individuals models but also behaviors not found in them, e.g. when a programmer switches between top-down and bottom-up code comprehension.

Figure 1 is a graphical representation of the overall model. Each rectangle represents a comprehension process within the model. Every process starts with documentation and continues by matching documents to accumulated knowledge. All three models have short-term memory components and differ by level of abstraction. Chunking is the process of taking lower-level abstractions and collapsing them into a higher-level description or label. A chunk is knowledge structure that contains various levels of abstractions of lower-level information. Both the program and situation model have a chunking component as part of the process. Chunking results in the storage of the higher-level representation into long-term memory. The middle *knowledge base* contains examples of the types of knowledge associated with each of the three models.

Various aspects of this model were confirmed in prior studies. von Mayrhauser and Vans (1993*b*) showed for one enhancement task that the software engineer switched between all model components of the integrated model and reported actions occurring at all three levels of the model. von Mayrhauser and Vans (1993*a*) extended these results to include a debugging task. It also analysed for detailed action types. Both interpret observations in terms of useful tool capabilities. von Mayrhauser & Vans, (1994, 1996*b*, *c*)

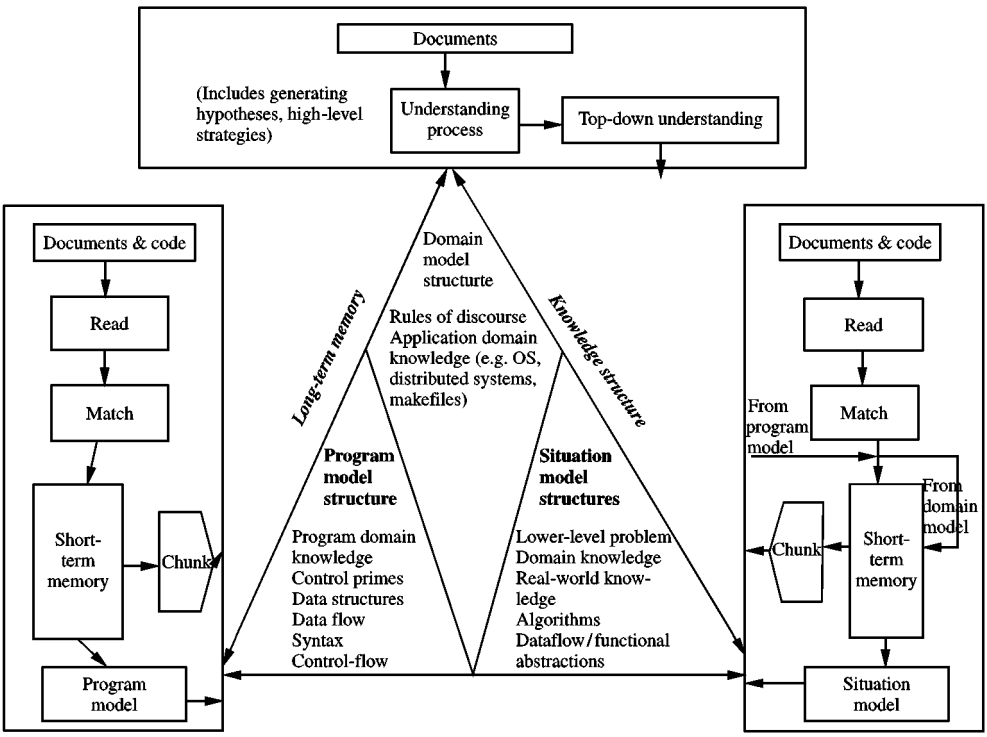

FIGURE 1. Integrated code comprehension model.

investigated whether observations could confirm the processes stipulated in the model. von Mayrhauser & Vans (1994, 1996*c*) report on the comprehension process of one subject who used a systematic understanding strategy. von Mayrhauser & Vans (1996*b*) report on the comprehension process related to an opportunistic strategy. It is structured around a hierarchy of goals, hypotheses and actions. von Mayrhauser & Vans (1997) report on comprehension behavior of two programmers during enhancement tasks. These results support the integrated model, the switching behavior between model components and the role of hypotheses in an opportunistic understanding process.

3. Design of study

We observed professional maintenance programmers working in the field on corrective maintenance. Software consisted of at least 40 000 lines of code. Each observation involved a *programming session*. Participants were asked to think aloud. We audio and/or video taped this as a thinking aloud report. Sessions were typically 2 hours long. As this is not enough to understand a large-scale software product, we classified participants by their degree of prior experience with the code.

The study ranked participants by levels of *expertise* and the amount of *accumulated knowledge* subjects had acquired prior to the start of each observation. Expertise

TABLE 1
Subject classification

Subjects	Task	Expertise	Accumulated knowledge
C1	Fix reported bug in terminal emulator program	Language expert (PASCAL), novice in application domain (communication protocols)	Some knowledge: file structure, call graph, requirements/design
C2	Understand bug in software project management program	Language novice (expert in Microsoft Windows and C, but not XT Intrinscis); domain expert (software project management software)	None
C3	Fix reported bug in operating system's kernel code	Language novice (expert in C, but not PASCAL); domain expert (OS kernel)	Little (file structure, call graph)
C4	Track down bug in client/server code	Language novice (expert in C, but not PASCAL); domain expert (client–server, OS)	Significant (prior maintenance tasks on same software)

distinguishes between programming knowledge (language and platform skills and application domain knowledge. This is the same classification used in earlier studies (von Mayrhauser & Vans, 1995*a*, 1996*c*). Shaft and Vessey (1996) confirmed the need to distinguish between programming knowledge and application domain knowledge because comprehension processes differ depending on the amount of programming and application domain knowledge. The amount of accumulated knowledge is likely to affect the work process as well. For example, someone who is already quite familiar with the code itself can be expected to refer to existing knowledge more often rather than have to acquire it. Table 1 describes the four subjects and the software they worked on.

The software and the specific assignment are representative of corrective maintenance in industry. It addresses situations in which maintenance programmers find themselves on occasion being assigned software in a new application domain or having to work with software in a "new" language or platform, but in an application domain where the programmer has expertise.

The subjects represent a sampling of application domains, prior work with the code and programmer experience in domain and language. While a case study like this is limited in its generality, the results provide useful insight and can serve as starting points for further investigations. The work presented here adds to prior observations reported in von Mayrhauser & Vans (1996*b*) in which a similar analysis was done for someone who was porting programs, rather than doing corrective maintenance. Further, the work reported here adds to prior observations of other types of maintenance tasks reported in von Mayrhauser & Vans (1993*a*, *b*, 1995*a*, 1996*a*, *c*, 1997). Our objectives were to analyse the observations for characteristic activities and behavior. The analysis also identified information needs. We used protocol analysis for this purpose.

3.1. PROTOCOL ANALYSIS

Protocol analysis proceeded in the same steps used in prior analyses (von Mayrhauser & Vans, 1993*a*, *b*, 1995*a*, 1996*a*, *c*, 1997). Protocol analysis is used for analysing observational data. Think-aloud reports of subjects are transcribed and classified using *a priori* categories determined from the literature and pilot observations. Each statement in the transcript is *encoded* as one of these *a priori* categories. We began with a list of expected actions Vessey (1985) and searched for them in the transcripts of the protocols. We also analysed for possible new action types. The analysis proceeds from identifying single actions of various types to determining action sequences and extracting cognition processes and strategies.

3.1.1. Actions

The first analysis on the protocols involved *enumeration of action types* as they relate to the integrated comprehension model (von Mayrhauser & Vans, 1995*b*). Action types classify programmer activities, both implicit and explicit. For example, we expect to find maintenance engineers generating hypotheses and reading code during maintenance. Each type is given an identifying code. Table 5 in the Results section lists all action types and their corresponding codes (see Appendix A for a definition of each action type). The results of this analysis are traces of action types as they occurred in the protocol. Summary data is then computed as frequencies for individual action types as well as cumulative frequencies of actions per model component.

3.1.1.1. Segmentation & information needs. The next step in the analysis combines *segmentation* of the protocols and identification of information and knowledge items. Segmentation classifies action types into those involving the domain, situation or program model that can be thought of in terms of different levels of abstraction in the mental model. For example, "examine data-structure" is a type of action. It is an action type related to program code and thus at the program model level. *Information needs* are information and knowledge items that support successful completion of maintenance tasks. Information needs are determined from protocols directly (see Table 3) or through inference. Tables 2 and 3 contain example results of action type and information needs analysis, with the tags and example protocol utterances.

TABLE 2
Example protocol analysis—action types (von Mayrhauser & Vans, 1993a, 1996c)

Analysis type	Tag	Action type	Example protocol
Action-type classification	Sys8	Generate hypothesis (program model)	"..and my assumption is that nil with a little *n* and nil with a big *N* are equivalent at the moment."
	Sys7	Chunk & store knowledge (program model)	"So clearly what this does is just flip a logical flag"

TABLE 3
Example protocol analysis—information needs (von Mayrhauser & Vans, 1993a, 1996c)

Analysis type	Tag	Information need classified as	Example protocol
Identifying inforamtion needs	I3	Code block boundaries	"Okay well, assuming that the … um indentation is accurate…I would guess that this is really for, um, there must be a FOR statement that this is the end of but I don't know where that FOR statement might come from. I don't see…"
	I61	Data structure tied to concepts in the domain	"And this looks like some X.25 structure"

3.1.1.2. Process analysis. Process analysis determines the nature of actions over time. Each action is classified by level of abstraction (program, situation or domain level). They can then be plotted as a function of "time" (in terms of numbers of actions). Graphs illustrate both, how long (in terms of actions) a programmer spends in each model, as well as the frequency of switches and whether switching is fairly unidirectional (top-down or bottom-up) or not. Switching frequencies and action traces are also examined statistically. A dependency analysis (Howe & Cohen, 1994) between pairs of actions identifies statistically significant patterns of switching behavior. Beyond that, we perform state transition dependency detection (STDD) (Howe & Somlo, 1997). STDD automatically constructs state transition diagrams for discrete event sequences. It works by consistently combining statistically significant short patterns into a larger unified model that illustrates the interrelationships.

3.1.2. Hypothesis analysis

Hypotheses are associated with model components. They are initially identified during action-type analysis by tagging generation, confirmation or failure of each hypothesis. Similar to action-type analysis, each statement identified as a hypothesis in the transcript is *encoded* as one of an *a priori* category (see Appendix B for a complete list and definition of hypothesis types). Each hypothesis is then classified by how it was stated in the protocol. E.g., "warnings are related, we believe, to the resources either not being installed properly or not being compatible with this system", is a hypothesis about the cause of buggy behavior. We classified the *type* of each hypothesis according to Letovsky's taxonomy into *what*, *why* or *how* hypotheses (Letovsky, 1986). Similar hypotheses are grouped into classes (e.g. hypotheses about program function).

We also follow each hypothesis through the protocol until it is resolved through confirmation, failure or abandonment. A hypothesis is verbally confirmed or rejected (i.e. fails). Hypothesis abandonment is explicit or implicit. Explicit abandonment occurs when the programmer decides it is not relevant or will be too much trouble to verify. Implicit abandonment occurs when the hypothesis is stated but the programmer

TABLE 4
Action-types by model—totals & frequencies

Subject code	Domain model	Program & situation model	Situation model	Program model	Total actions
C1	50	317	170	147	367
Corrective	14%	86%	46%	40%	
C2	130	258	101	157	388
Corrective	34%	66%	26%	40%	
C3	127	214	43	171	341
Corrective	37%	63%	13%	50%	
C4	181	208	117	91	389
Corrective	47%	53%	30%	23%	
Total	488	997	431	566	1485
Corrective	33%	67%	29%	38%	

never returns to it. The hypothesis was either forgotton or dismissed without verbal confirmation.

We counted frequencies of hypotheses at each model level, constructed contingency tables and analysed the table using a χ^2 test to determine whether the subjects show statistically significant differences in making hypotheses.

4. Results

4.1. PROGRAMMER ACTIONS

Table 4 shows how often the subjects perform actions at the three levels of abstraction defined in the integrated model. Percentages indicate relative frequency of these actions for each subject. The table also contains a column for combined program and situation model references, roughly corresponding to Pennington's (1987*a*) comprehension model. We wanted to identify patterns based on differences between Pennington's bottom-up model and the domain model.

Both C2 and C3 had a similar distribution of domain and combined program and situation model actions. Each had twice as many references to the combined program and situation models as to the domain model. Both were domain experts but had either very little or no prior experience with the code. We hypothesize that the lack of experience with the code and the language can cause this behavior of concentrating on lower levels of abstraction. This agrees with Pennington's (1987*b*) results that programmers tend to build a program model first when they are unfamiliar with the code.

C1's model preferences were different. This programmer was a language expert, but had very little experience in the domain. He had some prior experience with the code, including familiarity with the file structure, program call graph and requirements and design documents. Only 14% of his model references were in the domain model. We believe that lack of domain experience drives his behavior. Without the domain

knowledge, the programmer stays within the program and situation models until that experience is acquired. With language expertise, the programmer can concentrate on understanding the code at the program model level and use the situation model as a higher-level abstraction until he has acquired enough knowledge to make connections to the domain model.

The last subject, C4, had both domain experience and significant experience with the code. He had an almost equal distribution of domain and combined program and situation model references. This adds support to the hypothesis that domain expertise and experience with the code affects the ability to make connections between all three model levels. The distribution between the program and situation models was almost equal. We conjecture that both the domain expertise and the significant prior experience allowed this subject to make more use of the domain model.

Table 5 shows each action type by model for the subjects. The first column shows the code used in the protocols to identify the action. The second column contains a description of the action. The next four columns report the number of each action by subject. The last column lists cumulative actions for all subjects.

TABLE 5
Action types—corrective maintenance

Code	Action type	C1	C2	C3	C4	Total
OP1	Gain high-level program overview	6	16	0	5	27
OP2	Determine next program segment to examine	7	3	23	12	45
OP3	Generate/revise hypothesis re: functionality	9	18	26	24	77
OP4	Determine relevance of program segment	1	8	1	8	18
OP5	Determine if program segment needs detail understanding	0	0	0	1	1
OP6	Determine understanding strategy	8	12	14	9	43
OP7	Investigate oversight	0	1	0	1	2
OP8	Failed hypothesis	2	1	1	0	4
OP9	Mental simulation	0	0	0	1	1
OP11	High-level change plan/alternatives	0	7	0	1	8
OP12	Observe buggy behavior	0	0	0	3	3
OP13	Study/initiate program execution	0	3	0	20	23
OP14	Compare program segments	0	2	1	0	3
OP15	Generate questions	1	4	4	5	14
OP16	Answer questions	0	1	2	1	4
OP17	Chunk & store knowledge	4	9	11	5	29
OP18	Change directions	0	2	0	0	2
OP20	Generate task	4	9	0	36	49
OPCONF	Confirmed hypothesis	2	10	6	10	28
OPKNOW	Use of domain knowledge	6	24	38	39	107
Total	Domain model actions	50	130	127	181	488
SIT1	Gain situation knowledge	38	10	0	1	49
SIT2	Develop questions	13	4	0	7	24
SIT3	Determine answers to questions	6	1	0	1	8

TABLE 5 (Continued)

Code	Action type	C1	C2	C3	C4	Total
SIT4	Chunk & store	41	29	13	24	107
SIT5	Determine relevance of situation knowledge	5	4	2	4	15
SIT6	Determine next information to be gained	6	2	0	7	15
SIT7	Generate hypothesis	17	18	7	22	64
SIT8	Determine understanding strategy	5	2	2	5	14
SIT10	Failed hypothesis	1	0	0	1	2
SIT11	Mental simulation	0	2	0	2	4
SIT12	Compare functionality of 2 versions	0	2	0	0	2
SITCONF	Confirmed hypothesis	9	10	1	15	35
SITKNOW	Use of situation model knowledge	29	17	18	28	92
Total	Situation model actions	170	101	43	117	431
SYS1	Read introdutory code comments/related documents	4	15	7	1	27
SYS2	Determine next program segment to examine	15	7	3	1	26
SYS3	Examine next module in sequence	21	29	43	9	102
SYS4	Examine next module in control flow	1	2	0	13	16
SYS5	Examine data structs & definitions	4	0	0	1	5
SYS7	Chunk & store knowledge	21	23	51	20	115
SYS8	Generate hypothesis	24	12	18	11	65
SYS9	Construct call tree	3	0	0	0	3
SYS10	Determine understanding strategy	19	14	9	4	46
SYS11	Generate new task	0	13	0	8	21
SYS12	Generate question	0	9	1	1	11
SYS13	Determine if looking at right code	0	3	0	4	7
SYS14	Change direction	0	1	0	0	1
SYS15	Generate/consider different code changes	0	13	0	0	13
SYS16	Answer question	0	0	1	0	1
SYS17	Add/alter code	0	2	0	0	2
SYS19	Failed hypothesis	2	2	2	3	9
SYS21	Mental simulation	5	0	3	1	9
SYS23	Search for var definitions/use	0	1	3	0	4
SYS24	Search for block begin/end	2	0	0	0	2
SYSCONF	Confirmed hypothesis	2	6	9	3	20
SYSKNOW	Use of program model knowledge	24	5	21	11	61
Total	Program model actions	147	157	171	91	566

For domain model building, using domain knowledge is clearly the most frequent action (107). Domain knowledge was used 23% of the time during domain model construction. This supports our hypothesis that when programmers have domain knowledge they use it frequently. Three of the four subjects were domain experts. Together, C2, C3, and C4 made 94% of the references to domain knowledge. The second most important action type for domain model building is generating or revising hypotheses (OP3 = 77). Using hypotheses for building a domain model is a feature of Brook's theory of program comprehension (Brooks, 1983).

For situation model building, chunking and storing acquired information (SIT4 = 107) was most frequent, confirming Pennington's model. The subject with no domain experience and some prior exposure to the code had significantly more chunk and store actions than the rest of the subjects. This was probably due to this programmer's lack of familiarity with the domain, coupled with a high level of expertise (and thus feeling comfortable) with the programming language in which the software was written. Similar to the domain level, both use of situation model knowledge (92) and generating hypotheses (SIT7 = 64) are important. This not only confirms Pennington's model, but also supports hypotheses as major drivers of comprehension.

At the program model level, chunking and storing knowledge (SYS7 = 115) is the most frequent, confirming Pennington's theory that programmers build higher levels of abstraction from lower-level information. Examining code in sequence (SYS3 = 102) and generating hypotheses (SYS8 = 65) are the next two most frequent actions, demonstrating the importance of hypotheses in understanding code. Use of program knowledge has the fourth highest frequency (61).

The use of knowledge and generating hypotheses are important activities for programmers. In all three model components of the integrated model, these actions types ranked in the top four most-frequent actions. Chunking and storing information is the most important action for both the program and situation model levels. This is an indication of the building of knowledge. That reading code in sequence happened so often indicates a systematic strategy (Littman *et al.*, 1986).

The types and distribution of actions between the models tells us something about the level of abstraction at which programmers prefer to work. The integrated model defines mental model construction as the abstraction of lower-level information into higher-level abstractions or by decomposing the higher levels into lower levels. We can see this by looking at how often programmers switch between the levels and what the preferred models are between which the switch occurs.

4.2. PROCESSES

Switches occur between all three models (domain, program and situation models). Table 6 summarizes switches between models during the corrective maintenance task.

TABLE 6
Action switches—absolute & frequency

Task	Model	Model switches		
		Domain model	Situation model	Program model
Corrective (4 subjects) (Total switches = 562)	Domain	N/A	67 12%	93 17%
	Situation	86 15%	N/A	112 20%
	Program	75 13%	129 23%	N/A

The rows represent starting models and the columns represent ending models. Typically, switching between program and situation models happens because the engineer is trying to link a chunk of program code to an algorithmic description in the situation model. (A switch from program to situation model.) Alternatively, the programmer may be looking for a specific set of program statements to verify the existence of some functionally. (A switch from situation to program model.)

Switches during corrective maintenance occur slightly more often between program and situation models. For corrective maintenance, having that low-level information is important. It helps to more effectively track down defects and understand them. We are interested in understanding how expertise and amount of accumulated knowledge affect mental model construction. To see this, we looked at each subject's switching behavior, both in terms of how often they switched between models as well as the actual sequence of switches during the programming session. Figures 2–5 illustrate how these switches happen over time. These graphs are modeled after those found in Pennington, Lee and

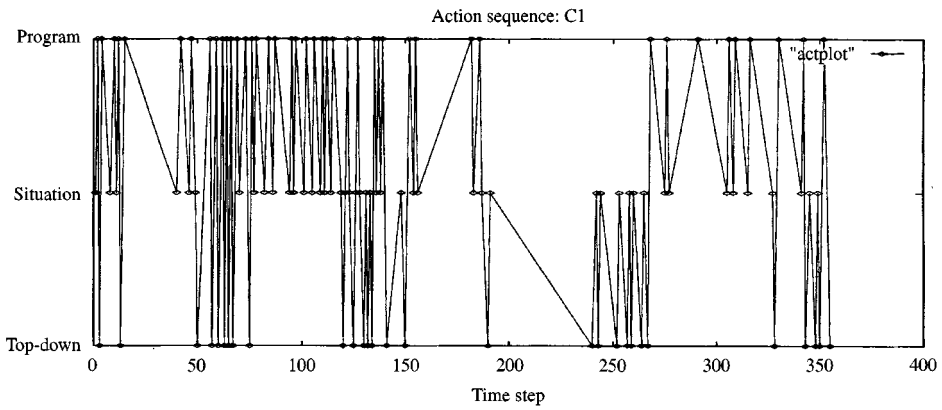

FIGURE 2. C1: fix reported bug—action sequence.

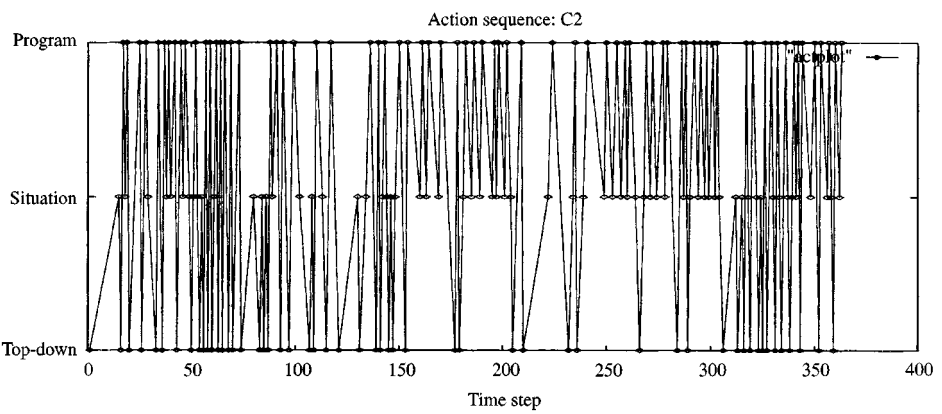

FIGURE 3. C2: understand bug—action sequence.

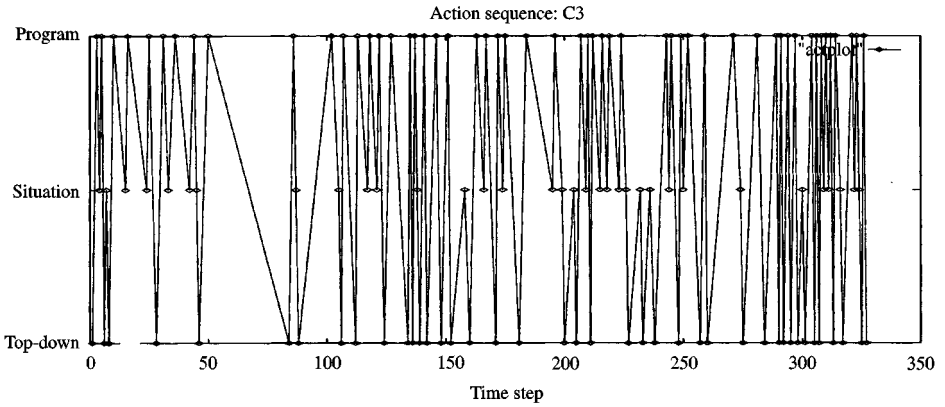

FIGURE 4. C3: fix reported bug—action sequence.

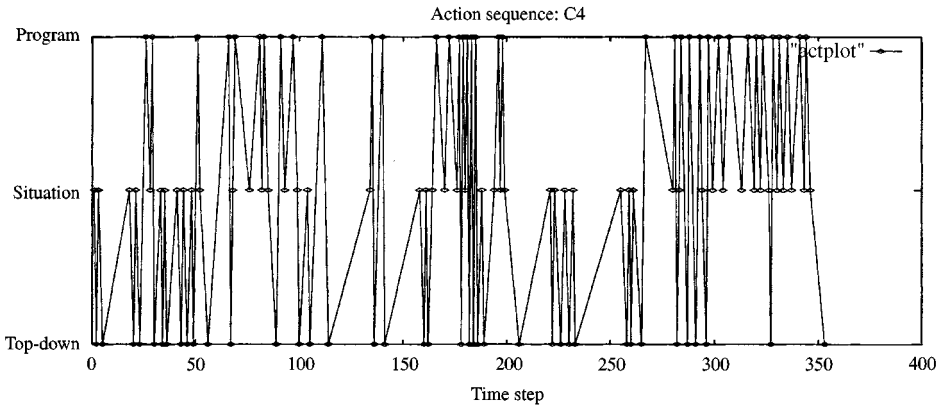

FIGURE 5. C4: track down bug—action sequence.

Rehder (1995) and Lee and Pennington (1994). The graphs show a line from one model component to another when the programmer switches from an action in one model to another. Thus the incline of the line represents how long a programmer stayed at a level before switching: a very steep line indicates few actions between switches, large number of actions between switches is indicated by a shallow incline or decline in the switch-line.

C1 switched 118 times. Switches between the situation and program model are more frequent than switches between either the domain and program models or the domain and situation models. We hypothesize that this is due to the subject's lack of domain knowledge. Figure 2 shows that this subject preferred to work at the situation and program model levels, switching quite frequently between the two and only occasionally switching to the domain model. The graph also shows that he spent more action time either in the program or situation models since the jumps into the domain model were quickly followed by a jump back into the program or situation model. Figure 2 illustrates

the hypothesis that programmers who are unfamiliar with the domain prefer to work at lower levels of abstraction.

C2 switched 192 times during the programming session. C2 switched slightly more often between the domain and program models than C1. However, switches between the program and situation model occurred at the highest frequency. Because C2 is knowledgeable about the application domain, he can make connections between the program and domain levels. We see similar behavior with C3. While C2 did not spend significant amount of time in the domain model before switching to another level, he spent more action time in the domain model than did C1. Because he was unfamiliar with the code, the frequent switching into the program model would indicate that he was trying to use his knowledge of the domain to understand the code. He also made use of the situation model, which could indicate that he used the situation model as an intermediate level bridge between the program and domain levels. The majority of the switches into the situation model occurred from the program model.

Similar to C2, C3 switched frequently between the program and domain models, but he did not use the situation model as a bridge from the domain to the program model as often as C2. Instead, he preferred to work at a particular model level, switching less frequently than C2 (C2 switched 192 times and C3 127 times).

Domain expertise may influence this programmer's ability to switch directly between the domain and program model levels. Accumulated knowledge may also play a part and drive this programmer's need to build a mental representation of the code in the most efficient manner. Having a domain mental representation allows the programmer to quickly isolate code that needs correction.

C4 switched 125 times between models. Switching is distributed pretty evenly among all three models except for direct switches between the domain and program model. They are lower (10% vs. 20%). We hypothesize that this is due to the amount of accumulated knowledge. Because C4 had been working with the code for a while, he already had mental representations at all three levels. The program understanding activities could be indicative of trying to fill in the holes at each level.

Figure 5 shows that he could switch between all three levels, preferring to concentrate on building specific levels of abstraction at various times during the session. For example, the first 50 or so actions alternated between the domain and situation model level activities, the next 50 between the program and situation model levels. Actions 100–150 are dominated by switches between the domain and program model levels. Then he repeats this pattern for a short time before concentrating on the domain and situation models for approximately 100 actions. The sequences of action switching lends support to the hypothesis that C4 was trying to complete mental model construction at all three levels of abstraction.

Next, we analyzed the action trace statistically (Howe & Cohen, 1994; Howe & Somlo, 1997). We did this to identify whether the sequences of actions by a single subject could have happened by chance or not, and, if they did not, to see with what probabilities individual subjects switched between model levels. Table 7 shows for all four subjects which action pairs (switches or transitions) are statistically significant (not likely to be due to chance). The first two columns list the action pairs by component model of the integrated comprehension model. The remaining columns report the significance level of the pairwise pattern for the four subjects, respectively. A blank cell indicates that this

TABLE 7
Dependency analysis results

Model action pair		Probabilities for programmer			
Model from	Model to	C1	C2	C3	C4
Domain	Domain	0.0000003	0.0000001	0.0000006	0.0000005
	situation	0.006368	0.050597	0.0011806	0.0000004
	program	0.12514	0.0000115	0.000001	0.000001
Situation	Domain	—	—	—	0.0000003
	situation	0.000001	—	—	0.0000002
	program	0.000001	—	—	—
Program	Domain	0.00291	0.0000007	0.0000014	0.0000006
	situation	0.0000009	—	0.0090065	—
	program	0.0000006	0.000378	0.0000003	0.0000008

pattern was not statistically significant and likely due to chance. The two subjects who are language novices and domain experts without much accumulated knowledge of the code, have fewer significant action pairs involving the situation model. By contrast, both the language expert and the subject who is familiar with the code through prior work with it show a richer switching behavior.

Next, we used STDD (Howe & Somlo, 1997) to construct a model of the transitions and switches between the domain, situation and program models. Figure 6 shows the results of this analysis. The nodes labeled OP represent the domain model, while the nodes labeled SIT and SYS represent the situation and program models, respectively. Transitions between states are labeled with transition probabilities that are statistically significant. STDD constructs a state transition diagram from statistically significant patterns found in the action trace. A threshold α indicates the maximum probability that a particular pattern might have been observed due to noise or chance; thus an estimated probability below α means that the pattern was *not* likely to have appeared due to noise. Rare patterns are likely to be significant; so, the diagram includes even low transition probabilities. The analysis was done using a threshold value of $\alpha = 0.15$ (so p must be <0.15 for the pattern to be included) for the dependencies in the action trace of length two.

Figure 5(a) shows C1's tendency towards bottom-up understanding in the high probabilities to stay at the program and situation model levels with switches between the two when enough knowledge has been accumulated. Figure 5(b) and (c) by contrast show the attempts by the domain experts/language novices to leverage their domain knowledge to index directly into the code. There is little use of the situation model. Figure 5(d) shows the behavior of the domain expert with significant accumulated knowledge about the software. The process is balanced between all three model levels with higher probabilities of staying at a given model, occasionally switching between model components as needed. The difference between C1 and C4 is in the much smaller amount of bottom-up understanding for C4 (as evidenced by significant transitions between the

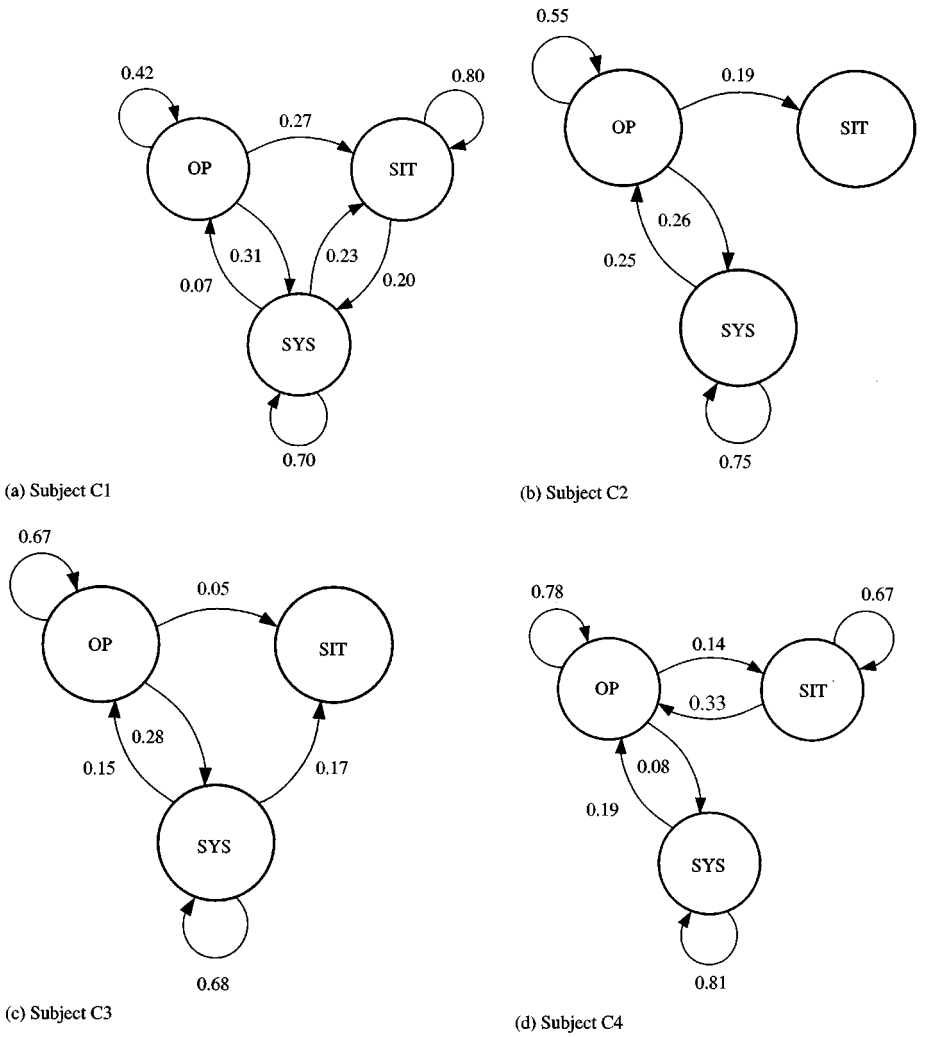

FIGURE 6. State transition diagrams.

program and the situation model in either direction). This analysis supports our earlier interpretation of the subjects' work process.

Comparing the four subjects, we offer the following interpretation regarding the effect of expertise and accumulated knowledge for corrective maintenance.

- Programmers with little experience in the domain, work at lower levels of abstraction until enough domain experience allows them to make connections from the code to higher levels of abstraction.
- Programmers with domain experience but little or no accumulated knowledge about the software also work at lower levels of abstraction, but use their knowledge of the

domain to make direct connections into the program model. They may also use the situation model as a bridge between the program and domain models. We saw a significant increase in the number of switches for subject C2, which could be an indication that smaller steps in comprehension are necessary until more experience with code is acquired.

- Programmers with domain expertise and significant experience with the software can make connections between all three levels of abstraction. They already have a good mental representation at all three levels and switch between levels when necessary to complete missing parts of the mental model.

4.3. TYPES OF HYPOTHESES

Table 8 lists all hypothesis types at each level of abstraction of the integrated comprehension model. The leftmost column identifies at which model level the hypothesis is made, the second column reflects the tag used to code the hypothesis type. The third column describes the theme of the hypothesis. The next column reports cumulative counts for each hypothesis type for all subjects combined. The last four columns state hypothesis-type frequencies for each subject individually. These hypothesis types are a refinement of hypothesis classes reported in von Mayrhauser and Vans (1995*b*).

Table 9 reports by subject and model level the number of hypotheses made. While the individual distribution of hypotheses over the three model levels varies, cumulatively they make almost the same number of hypotheses in each model (77 at the domain model level, 65 at the program level and 64 at the situation model level). Looking at Pennington's bottom-up understanding model as comprising the program- and situation model, the subjects make two-thirds of the hypotheses at that level. The most frequent hypotheses are about code correctness (OPH18-location/status/description/cause of error; SYSH16, code correctness, cause/location of error). At the program model level, the most frequent hypothesis made concerns statement execution order or program state. This maps well into current understanding of how corrective maintenance works. χ^2 test results show significant differences between subjects in terms of level of abstraction at which hypotheses are made. Based on prior process analysis, this is to be expected and likely driven by domain expertise vs. language expertise, as well as accumulated knowledge about the software.

The majority of hypotheses at the domain model level concerned location/status/description/cause of error (OPH18). This makes sense in light of their goal to fix a known defect.

The other three most frequent types of hypotheses are evenly divided between domain procedure/function concepts (OPH1), understanding the environment surrounding the system and the tools used (e.g. the debugger) (OPH9), and level & structure of code (OPH16). Code structure hypotheses are expected as it is necessary to understand code structure when trying to locate a bug. We surmise that domain concept hypotheses are frequent because three of the four subjects were domain experts and used their grasp of the domain to direct their understanding of the bug.

The highest percentage of program model hypotheses for corrective maintenance subjects concerned statement execution order (SYSH6). This appears to be task-related, because, when trying to locate a defect, understanding the order in which statements are

TABLE 8
Hypothesis-type frequencies

Model	Tag	Hypothesis type	Total refers	C1	C2	C3	C4
Top-down (domain) model	OPH18	Location/status/description/ cause of error	20	1		8	11
	OPH1	Domain procedures functionality/concepts	11	1	3	5	2
	OPH9	Permissions/environments set correctly/tool functionality	11		2		9
	OPH16	Level & structure of code/scope	11	5		6	
	OPH7	Number/type/existence/ location of libraries	5		4	1	
	OPH13	Number/type/location of file	5	1	2	1	1
	OPH6	Existence of specific functionality	3		3		
	OPH8	Program functions correctly	1				1
	OPH14	Available functionality	2			2	
	OPH2	Variable functionality/domain concepts	2	1	1		
	OPH3	Rules of discourse/expectations	1			1	
	OPH11	Comparison of functionality at high level	2		1	1	
	OPH17	Design decisions/modifications	2		2		
	OPH19	Current location	1			1	
Program model	SYSH6	Statement execution order/state	15	11	1	2	1
	SYSH16	Code correctness, cause/location of error	10	4		6	
	SYSH7	Variable value/defaults	8	1		2	5
	SYSH4	Variable structure	7		4		3
	SYSH10	Syntax meaning	7	5		2	
	SYSH2	Function/procedure function	6		2	3	1
	SYSH8	(Non-)Existence of construct (var/code)	3		2		1
	SYSH18	Location to add code/ alternatives	2	1	1		
	SYSH13	Code block/procedure comparison	2	2			
	SYSH20	Params/type definitions in procedures call	2			2	
	SYSH9	Variable/construct equivalency	1		1		
	SYSH1	Variable function	1		1		
	SYSH5	Location/type/existence of function call	1			1	
Situation model	SITH3	Function/procedure function, call function	25	8	7	3	7
	SITH7	Existence of functionality/ algorithm/variable	10	2	4	1	3

TABLE 8 (Continued)

Model	Tag	Hypothesis type	Total refers	C1	C2	C3	C4
	SITH2	Function/code block execution order/state	9	1	6	1	1
	SITH1	Variable function	7	2			5
	SITH5	Cause of buggy behavior	6	1		2	3
	SITH6	Comparison of terms/ acronyms/functionality	4	3	1		
	SITH8	Program function	2				2
	SITH4	Effect of running program	1				1

TABLE 9
Hypotheses by model—frequencies & percentages

	Top-down model	Program & situation model	Sit. model	Prog. model	All
C1	9 18%	41 82%	17 34%	24 48%	50
C2	18 37%	30 63%	18 37%	12 26%	48
C3	26 51%	25 49%	7 14%	18 35%	51
C4	24 42%	33 58%	22 39%	11 19%	57
Total	77	129	64	65	206
Chi-square results	$\chi^2 = 12.597$ $p < 0.005594$		$\chi^2 = 22.522$ $p < 0.000973$		

or could be executed is important. The second most frequent type is code correctness (SYSH16). Obviously, understanding whether the code is correct or not is an integral aspect of debugging. The third and fourth most important hypothesis types are variable value/defaults (SYSH7) and variable structure (SYSH4). Understanding variables and their values is essential when debugging code.

At the situation model level, the most frequent hypotheses concern functionality at the procedure level. The subjects seemed more concerned with functionality at the situation model level that at the program model level. Perhaps this is due to spending more time looking at statement execution order and variable values at the program model and abstracting functionality to the situation model. Hypotheses about variable and procedure functionality at the program model level is tied to language meaning; for example,

TABLE 10
Distribution of hypotheses categories

	What	How	Why	Confirmed	Abandon	Fail
C1	23	25	2	13	33	4
C2	31	13	4	26	18	4
C3	29	22	0	16	24	11
C4	29	25	3	28	18	11
Total	112	85	9	83	93	30
Chi-square results		$\chi^2 = 9.2267$ $p < 0.161225$			$\chi^2 = 20.0096$ $p < 0.002662$	

"variable 'i' is used as a looping mechanism". The second most frequent hypothesis type, existence of function/algorithm/variable (SITH7) makes sense. During the understanding of a defect, it is important to determine whether there is a program construct critical to code correctness, as its absence may be a contributor to the defect.

Next, consider the nature of the hypotheses made (*what*, *how*, *why* hypotheses and whether they were confirmed, abandoned or failed). Table 10 gives the total number of hypotheses in each of these categories for each subject and cumulatively for all subjects.

The majority of hypotheses were of type *what*. Letovsky (1986) defines *what* hypotheses as those that conjecture about what something is or does. There were far fewer (85) *how* hypotheses (conjectures about the way something is accomplished) and only 9 *why* hypothesis (conjectures about the objective of an action or design choice). The subjects had almost no *why* hypotheses. This may be due to their task (corrective maintenance) or to philosophy or attitude (e.g. "I only need to fix this, I don't care why the developers did this a certain way"). If this is indeed the case, such an attitude could be driven by (1) the lack of information about design or programming rationale and the difficulty in recreating it or (2) a lack of interest. In either case, not knowing why important decisions were made can lead to bug fixes that severely degrade a product. Thus the small number of "why" hypotheses is of some concern. A χ^2 test indicates no significant differences between subjects in this regard, regardless of their expertise. Thus there is no statistical support for the following alternate interpretation: before one can ask about why something is done, one must first understand what is done and how it is done. If programmers do not understand enough about what and how something is implemented, they may not know enough about the software to make conjectures about why something is implemented. Thus one could explain the lack of more "why" hypotheses by surmising that programmers may just not know enough about the software to ask more "why" hypotheses.

4.4. HYPOTHESIS RESOLUTION

Hypotheses can be abandoned, confirmed or fail. The last three columns of Table 10 state the type of hypothesis resolution. There are about twice as many confirmed or

abandoned hypotheses as failed ones. Similar behavior was found for porting software (von Mayrhauser & Vans, 1996*b*). The large number of abandoned hypotheses is surprising in light of Brooks (1983), who found that programmers rarely abandon a hypothesis. Rather, they backtrack to find a different way to confirm hypotheses. While the large number of abandoned hypotheses appears to contradict Brooks (1983), there may be another explanation: Vessey (1985) found that experts are flexible in their approaches to comprehension. They are able to let go of questions and assumptions more easily than novices. While the four subjects were not experts in both language and domain, they were expert programmers and thus expected to abandon questionable hypotheses. One could argue that the large number of (later abandoned) hypotheses speaks to the flexibility in which the subject approaches the comprehension problem. This appears to confirm Vessey's findings. A χ^2 test reveals significant differences between subjects in how they resolve hypotheses. It is unclear whether this is due to individual differences or their expertise.

On average, the subjects each confirmed 20.75 hypotheses, abandoned 23.25, and refuted 7.5. This compares to 16 confirmed, 14 abandoned and 8 failed hypotheses when porting code (von Mayrhauser & Vans, 1996*b*). Thus, on average, corrective maintenance programmers appear to make more hypotheses (51) than those porting code (38). Most of the "extra" hypotheses are abandoned, a distinct difference between corrective maintenance vs. porting. It is possible that a lack of knowledge is intrinsic to the task of trying to find an (unknown) bug and that this is contributing to the larger number of abandoned hypotheses. The large number of abandoned hypotheses could also represent an "arsenal" strategy (von Mayrhauser & Vans, 1996*b*) where a large number of possible answers (hypotheses) are generated in relation to a goal or higher-level hypothesis and less promising or useful ones are abandoned quickly. Only the most promising are pursued further.

4.5. DYNAMIC RESOLUTION PROCESS

Next, we analysed relationships between hypotheses, how sequences of goals and hypotheses affect achieving the overall goal of the task and tried to identify an overall strategy of hypotheses generation and resolution. For each goal, the analysis excerpted associated hypotheses, supporting comprehension actions and any subsidiary goals, hypotheses and actions. The result of the analysis is represented in a Table. For space reasons we show detailed results for two subjects only (Table 11 for C1, Table 12 for C4). Each corrective maintenance activity starts with the overall goal of the task, leading to the primary hypotheses (leftmost column). The hypothesis in turn is decomposed into subsidiary hypotheses and goals (next column) which is decomposed into further subsidiary hypotheses and goals (columns further to the right). Each hypothesis entry also lists the number of actions associated with working on the hypothesis and how it was ultimately resolved. (A—abandoned, C—confirmed, F—failed).† Each table thus shows

†Hypothesis 1.2.3.1.1.1 in Table 12 is marked with an asterisk to indicate that it has two subsidiary hypotheses that did not fit into the table.

- 1.2.3.1.1.3.1 SITH7, no associated actions, was abandoned.
- 1.2.4.1.1.3.2 OPH18, 8 associated actions, was confirmed.

TABLE 11
C1: goal–hypothesis–action triads

Level 1	Hypotheses level 2	Hypotheses level 3	Hypotheses level 4	Hypotheses level 5	Hypotheses level 6
H1 SYSH18 (F)	1.1 SYSH18 4 actions (A)				
	G1.1 Understand diference between syntax & expecta-tions	1.1.1 SYSH10 3 actions (A) 1.1.2 SYSH10 5 actions (A) 1.1.3 SYSH10 2 actions (A) 1.1.4 SYSH10 no actions (A)	1.1.3.1 SYSH10 no actions (A)		
	G1.2 Understand surrounding code	1.2.1 SITH3 2 actions (C) 1.2.2 OPH16 2 actions (C) 1.2.3 SITH3 3 actions (C) 1.2.4 SITH3 1 action (C) 1.2.5 SYSH10 3 actions (A) 1.2.6 SYSH13 no actions (A) 1.2.7 SYSH13 1 action (F) 1.2.8 SITH6 1 action (C) 1.2.10 SITH5 1 action (A) G.1.2.1 Understand Acronym 1.2.12 SITH3 3 actions (C) 1.2.13 SITH1 4 actions (A) 1.2.14 SITH7 6 actions (C) 1.2.15 SYSH6 no actions (F)	1.2.2.1 OPH16 2 actions (F) 1.2.8.1 SITH3 1 action (C) 1.2.10.1 SITH1 0 actions (A) 1.2.11 SITH6 55 actions (C) 1.2.13.1 SYSH6 2 actions (A) 1.2.13.2 SYSH6 2 actions (A) 1.2.14.1. SITH7 no actions (A)	1.2.2.1.1. OPH16 5 actions (A) 1.2.11.1 OPH13 no actions (A) 1.2.11.2 SITH3 no actions (A) 1.2.11.3 OPH16 no actions (A) 1.2.11.4 SITH6 no actions (A)	1.2.2.1.1.1OPH16 no actions (A)

TABLE 11 (Continued)

Level 1	Hypotheses level 2	Hypotheses level 3	Hypotheses level 4	Hypotheses level 5	Hypotheses level 6
		1.2.16 SYSH6 1 action (C)	1.2.16.1 SYSH6 5 actions (C)	1.2.16.1.1 SYSH6 1 action (A) 1.2.16.1.2 SYSH6 2 actions (A) 1.2.16.1.3 SYSH6 1 action (A)	1.2.16.1.1.1 SITH2 no actions (A)
		1.2.17 SYSH6 2 actions (A)			
	G1.3 Determine if problem where originally thought	1.3.1 SYSH16 11 actions (A) 1.3.2 SYSH16 1 action (A)	1.3.2.1 SYSH6 no actions (A)		

TABLE 12
C4: goal–hypothesis–action triads

Level 1	Hypotheses level 2	Hypotheses level 3	Hypotheses level 4	Hypotheses level 5	Hypotheses level 6
H1 Something wrong with way addresses are handled in server	1.1 OPH13 3 actions (C)				
	1.2 OPH18 22 actions (C)	1.2.1 OPH1 3 actions (C)			
		1.2.2 SITH3 10 actions (C)	1.2.2.1 SITH3 1 action (C)		
		1.2.3 OPH18 2 actions (C)	1.2.3.1 OPH18 3 actions (C)	1.2.3.1.1 SITH5 9 actions (C)	1.2.3.1.1.1 OPH9 1 action (A) 1.2.3.1.1.3 SITH5* 22 actions (C)
	1.3. OPH18 1 action (A)				
G1.2 Understand why date is chopped off in variable	1.4 OPH18 104 actions (A)	1.4.1 SYSH4 no actions (A)			
		1.4.2 OPH18 1 action (A)			
		1.4.3 OPH18 1 action (A)			
		1.4.4 OPH18 no actions (A)			
		1.4.5 SITH1 4 actions (C)	1.4.5.1 SYSH4 no actions (F)		
			1.4.5.2 SITH4 3 actions (F)		

TABLE 12 (Continued)

Level 1	Hypotheses level 2	Hypotheses level 3	Hypotheses level 4	Hypotheses level 5	Hypotheses level 6
		1.4.6 SYSH6 no actions (A)			
		1.4.7 SYSH7 1 action (C)			
		1.4.8 OPH18 no actions (A)			
		1.4.9 SITH1 no actions (A)			
		1.4.10 SITH5 no actions (A)			
		1.4.11 OPH9 5 actions (F)	1.4.11.2 OPH9 no actions (F)		
			1.4.11.3 OPH1 1 action (F)		
		1.4.12 SITH8 2 actions (F)	1.4.12.1 OPH9 1 action (C)	1.4.12.1.1 SITH8 2 actions (C)	1.4.12.1.1.1 OPH9 no actions (C)
		1.4.13 OPH9 2 actions (F)	1.4.13.2 OPH9 4 actions (F)		
		1.4.14 OPH8 no actions (A)			
		1.4.15 SYSH7 7 actions (F)	1.4.15.1 SYSH8 1 action (F)	1.4.15.1.1 SYSH7 no actions (A)	
		1.4.16 SITH7 3 actions (C)			
		1.4.17 SYSH7 1 action (C)			
		1.4.18 SITH7 no actions (C)			
		1.4.19 SITH2 3 actions (F)	1.4.19.2 SITH3 no actions (A)		
		1.4.20 SYSH7 3 actions (F)			
		1.4.21 SITH3 3 actions (C)	1.4.21.1 SITH3 5 actions (C)	1.4.21.1.1 SYSH2 1 action (C)	no actions (C)
		1.4.22 SITH1 no actions (C)			
		1.4.23 SITH1 5 actions (C)			
		1.4.24 SITH1 1 actions (C)	1.4.24.1 SITH3 1 action (C)		
		1.4.25 SITH3 no actions (A)			

a hierarchy of goal–hypothesis–action triads (von Mayrhauser & Vans, 1996*b*). From the top down, Tables 11 and 12 show sequentially the process of corrective maintenance. Walking through the table depth-first (from left-to rightmost column) represents the subjects' work process over time.

C1 is a language expert, but domain novice. His approach proved ultimately unsuccessful, his primary hypothesis failed. Because of his lack of domain knowledge, he approached the problem from his angle of expertise: the language. Bottom-up comprehension and chunking and storing acquired information starts the session (hypothesis 1.1 and goal 1.1). The steps are small (few actions between hypotheses). As he acquires knowledge and abstracts it, hypotheses at the algorithmic and functional level become possible (the first 10 hypotheses related to goal 1.2). A subsidiary goal (1.2.11) leads to a long series of actions (55) until the hypothesis can be confirmed and the acronym is understood. Abandoned hypotheses tend to appear in clusters (e.g. 1.2.11.1–1.2.11.4) with few or no actions associated. They are usually in his area of expertise, pointing to "arsenal" behavior (von Mayrhauser & Vans, 1996*b*).

C4 is a language novice and domain expert. Not surprisingly, his approach to finding the bug is directed from the top down, starting with three domain level hypotheses as subsidiaries to the primary hypothesis. The steps (number of actions) at that level generally are larger (OPH18 shows 22 actions associated with it, OPH18 has 104). Most of C4's abandoned hypotheses are at the domain level, his area of strength (analogously, C1 abandons most of his hypotheses at the program level, C1's area of expertise). In both C1 and C4's case, hypotheses in areas outside of their expertise are not as readily abandoned. This confirms results by Vessey (1985), pointing to their inability to let go of questionable hypotheses in areas where they are novices. C4 starts "small-stepping" when working at the program level, where he lacks knowledge (language novice). This corresponds to the small steps of the domain novice C1 when working at the domain level.

4.6. INFORMATION NEEDS

Information needs are developed by analysing each action type for the kind of information needed as part of the task. The most frequent information needs are summarized in Table 13. The information needs table contains seven columns. The first provides a code for the information need. The second describes the information for which the software engineer is looking. The third lists cumulatively how often the subjects had a need for this information. The last four columns provide detailed results for C1, C2, C3 and C4 individually. These results help to illustrate the information software engineers need to debug software and to prioritize solutions based on the relative importance of this information.

Overall, the most important types of information needed during corrective maintenance are domain concept descriptions (I7), connected domain, program and situation model knowledge (I16), location and uses of identifiers (I4) and a list of browsed locations (I9). Domain concept descriptions include high-level information, for example, operating system concepts. Connected model knowledge is information that allows cross-referencing from one model to another. For example, if a chunk of code is labeled "sort", the label can be viewed as the connection between program and situation models. These two types of information are needed much more often than other types of information. This strengthens the hypothesis that programmers try to build different levels of abstract views of the program. Recalling recently browsed information (I9) is also important. The ability to scroll back and forth between data reduces cognitive overload. A common

TABLE 13
Corrective maintenance—information needs

Code	Information need	Subjects total	C1	C2	C3	C4
I7	Domain concept descriptions	41	14	18	2	7
I61	Connected domain-program-situation model knowledge	33	5	12	10	6
I4	Location and uses of identifiers	26	3	8	8	7
I9	List of browsed locations	17	2	7	8	0
I2	List of routines that call a specific routine	14	7	1	6	0
I14	Call graph display	12	3	4	5	0
I73	Bug behavior isolated	11	1	0	5	5
I43	A general classification of routines & functions so that if one is understood the rest in the group will be understood	11	4	6	1	0
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions	8	4	1	3	0
I22	History of past modifications	7	3	2	0	2
I24	List of executed statements & procedure calls, variable values	7	0	0	0	7
I66	Expected program state, e.g. expected variable values when procedure is called	6	2	0	0	4
I72	Good direction to follow given what is already known, possible program segments to examine	6	1	3	2	0

example occurs when a programmer skims data he does not fully understand and needs to revisit the information when other data triggers a recall and subsequent comprehension of the previously skimmed data.

C1 needed domain concept descriptions most often (I7 = 14). The second most frequent need was a list of routines that call a specific routine (I2 = 7). The third most frequent was connected information from each of the models into the other models (I61 = 5). The need for domain information and model connections may be due to the lack of C1's domain experience and accumulated knowledge. While he preferred to work at the program and situation model levels, there were times when he recognized the need to understand functionality at a higher level. The need for function call information could be influenced by the lack of accumulated knowledge. This type of information is particularly important during initial comprehension. It becomes less important once the programmer has worked with code for a while.

For C2, domain concept descriptions (I7 = 18), connected model information (I61 = 12), and location and uses of identifies (I4 = 8) were the top three information needs. C2 had no accumulated knowledge about the software. The difference between C2 and C1, however, is that C2 had domain knowledge. This knowledge can be used to search for expected code segments. C2 also needed the most information. He had 32% of

all corrective maintenance information needs. This may be due to his lack of accumulated knowledge about the code. The need for low-level information is probably due more to the debugging task than domain expertise or accumulated knowledge.

C3 needed connected model information (I61 = 10), location and uses of identifies (I4 = 8), list of recently browsed code locations (I9 = 8) and list of routines that call a specific routine (I2 = 6) most often. While C3 needed connected information, his need for domain concepts was significantly less than the other three subjects. Instead, he needed lower-level information and wanted to follow a thread of reasoning by keeping track of where he had been in the code. C3 had a little accumulated knowledge about the code, but similar to C1 he needed to know call information (I2) because he was not yet very familiar with the code.

Domain concept descriptions (I7 = 7), location and uses of identifiers (I4 = 7), control-flow graph (I68 = 7) and connected model information (I61 = 6) are needed most often by C4. This subject had the smallest number of information needs of all corrective maintenance subjects. This is probably because he had both domain expertise and a significant amount of knowledge about the software. He needed specific information on the code he was debugging, and this shows up in the types of information he required.

The need for understanding domain concepts and building the connections between the three model components can be driven by different goals. For programmers with domain experience the need for this type of information may be driven by wanting to build their mental representations by connecting the knowledge they have with specific code segments. On the other hand, those with no domain knowledge need this information so that they can understand functionality of the code. We also found that the subject with the most experience, C4, needed less information than the other three since he was using existing knowledge.

5. Conclusions

Corrective maintenance is a frequent activity during software evolution. We observed four experienced professional programmers while they were debugging software and analysed their behavior. The goal was to answer several questions about how programmers go about debugging software, their work process and their information needs. Answers can be summarized as follows.

1. *Actions.* Use of knowledge and generating hypotheses are important programmer actions when working at all levels of abstraction. At the lower levels, chunking and storing acquired information is also common.
2. *Process.* Little experience in the domain means that comprehension will occur at lower levels of abstraction until enough domain experience allows connections to be made from the code to higher levels of abstraction. This confirms Pennington's results. Having domain experience but little or no accumulated knowledge about the software will cause comprehension to occur at lower levels of abstraction, but will also allow more efficient use of the existing domain knowledge to make direct connections into the domain model. The situation model can more easily be used as a bridge between the program and domain models. Domain expertise and significant experience with

the software allows connections between all three levels of abstraction to be easily made. We showed this through statistical dependency analysis.

3. *Information needs.* Domain concepts and connected program, situation and domain model information is important during corrective maintenance. The reason we saw the need for domain concepts and connected model information so frequently could be because it is the type of information that is not easy to get from existing tools and technology. The usual scenario is that any information above the program model level has to be searched for and connections made manually. Expertise and accumulated knowledge can affect the types of information needed. We saw that the programmers with domain expertise and accumulated knowledge about the code looked for very specific types of information, such as statement execution order and definition and uses of variables.
4. Corrective maintenance programmers make hypotheses at the code, algorithmic and application domain level, indicating the need to understand software at all levels of abstraction and to cross-reference this knowledge when locating and removing bugs. In spite of their differences, the programmers did not show statistically significant differences in making *what*, *how* and *why* hypotheses.

   Most hypotheses are confirmed or abandoned. Far fewer fail. Hypotheses are abandoned for a variety of reasons; resolving them may take too long, or another approach may become more promising. The high number of abandoned hypotheses is likely related to the expertise of the subjects and to corrective maintenance (the uncertainty associated with finding and removing a bug).
5. Goal completion, except for some iteration at the action level, was sequential. All subjects followed an identifiable "path" very methodically. Actions in support of hypothesis resolution are generally of the type found in the episode level processes of von Mayrhauser and Vans (1996*c*). Specific actions deal with code level entities like variable behavior, with algorithmic questions and domain concepts. Observing the comprehension processes as a function of expertise in application domain and language/platform, effectiveness (and possibly efficiency) was compromised in areas where the programmers were relative novices. Then they behaved like novices (Vessey, 1985). Providing information that lets programmers start in their area of strength and cross-reference into relevant, related information at other levels in which their knowledge is weaker, would help knowledge acquisition and likely increase effectiveness and efficiency.

Some of our results show that results obtained in experimental settings using small programs scale up to realistic corrective maintenance scenarios on large-scale code. For example, we saw hypothesis-driven understanding (Brooks, 1983; Letovksy, 1986) bottom-up (Pennington, 1987*a*, *b*), top-down (Soloway & Ehrlick, 1984; Soloway *et al*., 1988) and mixed bottom-up/top-down understanding during parts of the task. We also saw novice vs. expert behavior similar to shaft and Vessey (1996) and the use of tracing variable values (Weiser, 1982), etc. For the most part, we did not see these behaviors in the exclusivity observed in the experiments on small-scale code. Our subjects showed a much wider range of behaviors. This should come as no surprise, since their tasks were more demanding due to the large-scale code they had to deal with. This may also explain the behaviors we observed that seemingly contradict results of small-scale experiments,

like the relatively large number of abandoned hypotheses in our observations vs. Brooks' (1983) study.

We consider these conclusions as working hypotheses rather than generally validated behavior. The sample of subjects was simply too small. These conclusions should be validated through further observations. Unfortunately, such observational studies are costly. Although all four subjects worked on very different applications, commonalities emerged. We offered these as results of the current study.

References

BROOKS, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies*, **18,** 543–554.

HOWE, A. & COHEN, P. (1994). Detecting and explaining dependencies in execution traces. In P. CHEESEMAN, & R. W. OLDFORD, Eds. *Selecting Models from Data: Artificial Intelligence and Statistics Vol. IV*, p. 161–182. New York: Springer.

HOWO, A. & SOMLO, G. (1997). Modeling discrete event sequences as state transition diagrams. In X. LIU, M. BERTHOLD & P. COHEN Eds. *Advances in Intelligent Data Analysis*. Berlin: Springer.

LEE, A. & PENNNGTON, N. (1994). The effects of paradigm on cognitive activities in design. *International Journal of Man–Machine Studies*. **40,** 577–601.

LETOVSKY, S. (1986). Cognitive processes in program comprehension. In SOLOWAY & IYENGAR, Eds., *Empirical Studies of Programmers*, pp. 58–79. Norwood, NJ: Ablex Publishing Corporation.

LITTMAN, D. C., PINTO, J., LETOVSKY, S. & SOLOWAY, E. (1986). Mental models and software maintenance. In SOLOWAY & IYENGAR, Eds. *Empirical Studies of Programmers*, pp. 80–98. Norwood, NJ: Ablex Publishing Corporation.

PENNINGTON, N. (1987*a*). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19,** 295–341.

PENNINGTON, N. (1987*b*). Comprehension strategies in programming. In OLSON SHEPPARD & SOLOWAY, Eds., *Empirical Studies of Programmers: Second Workshop*, pp. 100–112. Norwood, NJ: Ablex Publishing Corporation.

PENNINGTON, N., LEE, A. Y. & REHDER, B. Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human–Computer Interaction*, **10,** 171–226.

SHAFT, T. M. & VESSEY, I. (1996). Computer program comprehension processes: the effect of application domain knowledge. *Empirical Studies of Programmers: 6th Workshop*, Alexandria VA, pp. 277–278.

SOLOWAY, E. & EHRLICH, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **SE-10,** 595–609.

SOLOWAY, E. ADELSON, B. & EHRLICH, K. (1988). Knowledge and processes in the comprehension of computer programs. In M. CHI, R. GLASER & M. FARR, Eds. *The Nature of Expertise*, pp. 129–152. London: A Lawrence Erlbaum.

VESSEY, I. (1985). Expertise in debugging computer programs: a process analysis. *International Journal of Man–Machine Studies*, **23,** 459–494.

VON MAYRHAUSER, A. & VANS, A. (1993*a*) From program comprehension to tool requirements for an industrial environment. *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, July 1993, pp. 78–86.

VON MAYRHAUSER, A. & VANS, A. (1993*b*) . From code understanding needs to reverse engineering tool capabilities. *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE99)*, Singapore, July, 1993, pp. 230–239.

VON MAYRHAUSER, A. & VANS, A. (1994). Comprehension processes during large scale maintenance. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 39–48.

VON MAYRHAUSER, A. & VANS, A. (1995*a*). Industrial experience with an integrated code comprehension model. *IEE Software Engineering Journal*, 171–182.
VON MAYRHAUSER, A. & VANS, A. (1995*b*). Program understanding: models an experiments. In M. C. YOVITS & M. V. ZELKOWITZ, Eds. *Advances in Computers*, Vol. 40, pp. 1–38. New York: Academic Press, Inc.
VON MAYRHAUSER, A. & VANS, A. (1996*a*). On the role of program understanding in re-engineering tasks. *Proceedings of the 1996 IEEE Aerospace Applications Conference*, Snow-mass, February 1996, pp. 253–267.
VON MAYRHAUSER, A. & VANS, A. (1996*b*). On the role of hypotheses during opportunistic understanding while porting large scale code. *Proceedings of the 4th Workshop on Program Comprehension*, Berlin, March 1996, pp. 68–77.
VON MAYRHAUSER, A. & VANS, A. (1996*c*). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, **22,** 424–438.
VON MAYRHAUSER, A. & VANS, A. (1997). Program understanding behavior during enhancement of large-scale software. *Journal of Software Maintenance*, **9,** 299–327.
WEISER, J. (1982). Programmers use slices when debugging. *CACM* **25**(7)**,** 446–452.

Appendix A: Classification of comprehension model—action codes

Note: Only the action types that occurred in the corrective maintenance protocols are shown in Tables A1–A4. For example, SIT9 did not occur in the protocols and therefore does not appear in Table A2.

TABLE A1
Situation model

SIT1—Gain situation model knowledge Specific intentions to get a situation/application level understanding. Comments indicating that background information is sought. For example, studying a text book on the subject of the application area. These comments are usually in the form of reading aloud some documentation
SIT2—Develop questions Questions or inquiries about the application area or situation. These questions are either immediately pursued or noted for later investigation. For example, "I wonder if they used the ISIS protocol for switcher communication?"
SIT3—Determine answers to questions Verbalized answers to stated questions, either immediately preceding the verbalization of the question, or if it directly answers a question that can be found in the protocol at an earlier time. Following the example question under SIT2 above, the statement "YUP! IT DID" would qualify as an SIT3 statement
SIT4—Chunk & store Comments summarising actions in the situation model. These typically involve statements about related real-world concepts, for example, the concept the program is attempting to simulate. Key words an phrase include "*So*, *thus*, *OK*"
SIT5—Determine relevance of situation knowledge Statements indicating a decision to understand/not understand a specific concept in more depth
SIT6—Determine next info to be gained Comments indicating the programmer is looking for a clue to help determine what information to gain next that might help in the comprehension process

TABLE A1 (Continued)

SIT7—Generate or revise hypothesis
Any time a comment about the situation or functionality using situation model language is unsure, assumed or not verified the comment is classified as an hypothesis. Key words and phrases include: "*Probably*", "*I guess*", "*I assume*", "*It could be*", "*I wonder*", "*I think*," "*I believe*", "*It seems*"

SIT8—Determine understanding strategy
Comments regarding how the programmer will or would have approached understanding. This is similar to SYS10 and OP6, except statements address understanding in the situation model. For example, "First I need to understand the change in inventory calculation, then … "

SIT10—Failed hypothesis
Comments indicating that something that was thought to be the case has been disproved or abandoned. These comments are about the real world/situation level related to the program. For example, "I was wrong, there are not just two protocol strategies there's a whole multitude of them". See situation knowledge below

SIT11—Mental simulation
Comments in situation model language indicating the programmer is mentally simulating program behavior. For example, "But, since we're only adding the new part inventory, then we need to create a new part number, then add the part number to the order database … "

SIT12—Compare functionality between two versions
Comments describing some kind of comparison of functionality at the situation level. This does not include direct comparisons of code, for example, results of the *diff* command. Comments that indicate that some concept should be/is being examined for use is a candidate for this category

SITCONF—Confirmed hypothesis
Statements indicating that a previously stated situation model hypothesis has been confirmed

SITKNOW—Use of situation model knowledge
Statements expressing information in the real world. For example, statements about operating system functionality or application area. This knowledge is typically associated with what is accomplished not HOW it is accomplished. (This is program model knowledge.)

TABLE A2
Domain model

OP1—Gain high-level overview of program
Specific intentions to get a high-level understanding. Comments indicating that only a high-level understanding is sought. For example, "I see how this program segment fits into the whole"

OP2—Examine/search for program information
Comments indicating that particular documents related to the program or information within specific documents are sought and read. Documents such as maintenance manuals, design documents, user manuals are included

OP3—Generate or revise hypothesis about functionality
Anytime a comment about the program in high-level terms in unsure, assumed or not verified the comment is classified as an hypothesis; e.g. "there is an initialization routine which I assume is called from …." Key words and phrases include: "*Probably*", "*I guess*", "*I assume*", "*It could be*", "*I wonder*", "*I think*," "*I believe*", "*It seems*"

OP4—Determine relevance of program segment
Comments indicating that the programmer is trying to decide if the segment being looked at is relevant to the current understanding task. Also included are comments that indicate that this decision has already been made; e.g. "I'm not going to worry about that now" or "we don't care about that"

TABLE A2 (Continued)

OP5—Determine if this program segment needs detailed understanding
Similar to OP4. May be no difference since no protocol segments have been classified as such

OP6—Determine understanding strategy
Comments regarding how the programmer will or would have approached understanding at a high level; e.g. "First I will read the Requirements Documents, then …."

OP7—Investigate oversight
Comments about overlooking some aspect of the code; e.g. forgetting to add some functionality or a whole routine

OP8—Failed hypothesis
Comments indicating that something that was thought to be the case has been disproved or abandoned. These comments are about the high-level structure of the program. For example, "I was wrong, there are not just two libraries there's whole multitude of libraries."

OP9—Mental simulation
Comments that indicate the programmer is simulating (control-flow or data flow) at a functional level

OP11—High-level change plan/alternatives
Descriptions of changes to code or alternatives being considered at a high level. For example, "I will add the changes to the make file and then delete the print function." Or, "I could copy some files and then remove the directory containing the print function, or just leave things as they stand now."

OP12—Observe buggy behavior
Comments indicating that program execution behavior is being observed and that it is not correct or expected. For example, "I got an error, normally this would not take this long if it had been doing shared library versions."

OP13—Study/initiate program behavior
Description of steps taken to initiate program execution or indicating that program execution is being observed. Comments describing the intention of executing commands or comments describing evidence of program execution

OP14—Compare program segments
Comments describing some kind of comparison at the program/application level in terms of functionality. This does not include direct comparisons of code, e.g. results of the diff command. Comments that indicate a similar program should be/is being examined for reuse at any level, e.g., design, specification, etc. Comments that include steps necessary to compare program segments

OP15—Generate questions
Verbalized high-level questions that are immediately pursued or noted for later investigation. These questions exist at the specification, design, application level. For example, "Did the program install the correct application defaults?"

OP16—Answer question
Verbalized answers to stated questions, immediately preceding the verbalization of the question, or directly answering a question that can be found in the protocol at an earlier time. Following the example question under OP15 above, the statement "YUP! IT DID" would qualify as an OP16 statement

OP17—Chunk & store knowledge
Very high-level statements indicating that some result has been realized. These involve statements about the program or document being studied AS A WHOLE. For example, "So, this program seems to work."

TABLE A2 (Continued)

OP18—Change directions
Comments that indicate that a different direction is to be/is being followed. For example, "I am going to try something completely different, I am going to copy these files over here instead."

OP20—Generate task
Comments describing additional tasks that are discovered and need doing some time in the future. Tasks involve high-level tasks e.g. reading a particular document. Most tasks referred to will occur sometime beyond the current programming session, unless a discovery is made and the task needs immediate attention

OPCONF—Confirmed hypothesis
Statements indicating that a previously stated domain model hypothesis has been confirmed.

OPKNOW—Use of domain knowledge
Comments indicating that knowledge acquired in the past before or during the programming session is used in reasoning. Include comments about previous sessions, e.g. reading documentation. Statements using knowledge acquired in the on-going program session, e.g. knowledge that is chunked and stored. Domain knowledge includes call-graph information, overall understanding how the program/program segment works

TABLE A3
Program model—part 1

SYS1—Read introductory code comments
Statements that are word-for-word reading aloud of comments in the code. This usually takes the form of a preceding statement such as "the comments say …." or "I'm reading the comments and it says … "

SYS2—Determine next program segment to examine
Any comment suggesting that a particular piece of code is sought after or comments indicating the programmer is looking for something to analyse. Key words and phrases include: '*I have to go find …*", "*I'm looking for …* "

SYS3—Examine next module in sequence
Comments reflecting code that is being read, word for word or in micro/macro-structure form or a description of what is being read using program model language. For example, the actual declaration of a variable

SYS4—Examine next module in control-flow
Comments reflecting code that is being read, word for word or in micro/macro-structure form or a description of what is being read using program model language. The difference between this task and examining a module in sequence is that control-flow is followed. For example, " … then within PS-RECEIVE this..", or "SET-SCREEN-MODIFIED is a sub-procedure.."

SYS5—Examine data structures and definitions
Comments suggesting that a data structure is being examined. For example, "Because STACK is a fairly generic name. … ok here's the definition of the variable, so it's *structure STATE*. So let me see if I can find a definition for *structure STATE*. And here it is."

SYS7—Chunk & store
Summarizing comments about code structure and the programming language. For example, "So, DO-ERASE-ALL-UNPROTECTED is called based upon ERASE-ALL-UNPROTECTED command." Keywords include *So*, *OK*, *Thus*

SYS8—Generate or revise hypothesis
Anytime a comment about program code is unsure, assumed or not verified the comment is classified as an hypothesis. Key words and phrases include: "*Probably*", "*I guess*", "*I assume*", "*It could be*", "*I wonder*", "*I think*," "*I believe*", "*It seems*"

TABLE A3 (Continued)

SYS9—Construct call tree
Comments about construction of where a particular procedure or variable is called form. For example, "I'm making a note of where SET-SCREEN-MODIFIED is referenced at. Or actually where it's called form."

SYS10—Determine understanding strategy
Comments regarding how the programmer will or would have approached understanding. For example, "by the way as odd as it may seem that actually is a determining factor as to how I would have approached this problem. When I originally looked at SET-SCREEN-MODIFIED I found out that it is only called from six locations. So I can manually do this for a relatively short period of time. … If there were 50 calls here I would not be doing this for the 50 calls."

SYS11—Generate new task
Comments indicating additional tasks have been generated or discovered as a result of comprehension activities. For example, "Oh, I'm going to have to go back in later and add some code to ….." Most tasks referred will occur sometime beyond the current programming session, unless a discovery is made and the task needs immediate attention

SYS12—Generate question
Verbalized questions about the code that are either immediately pursued or noted for later investigation. Specific questions as well as questions expressed as specific information needed. For example, "Let's see whether it runs over here on my HPUX workstation."

SYS13—Determine if looking at correct code
Explicit comments about finding the correct code or not. For example, "I am not looking at the correct code here."

SYS14—Change direction
Comments indicating the programmer is taking a different direction. For example, a different section of code will be examined because the current code is not supplying the information sought after

SYS15—Generate/consider alternative code changes
Specific comments indicating that code will be physically changed or the consideration of the effects of different code changes. For example, "Did the variable DATA get set?"

SYS16—Answer question
Comments that answer a specific question that was asked earlier about the code, either immediately preceding the verbalization of the question, or if it directly answers a question that can be found in the protocol at an earlier time. Following the example question under SYS15 above, the statement "YUP! IT DID" would qualify as an SYS16 statement. Also includes a simple Yes or No right after the question was asked

SYS17—Add/alter code
Comments describing the code change as it is happening, or code added in the immediate past

SYS19—Failed hypothesis
Comments indicating that something that was thought to be the case has been disproved or abandoned. These comments are specifically about statements in the code. See program model knowledge below

SYS21—Mental simulation
Comments in program model language indicating the programmer is mentally simulating program behavior. For example, "But, since we're only processing the first two arguments, then we leave this procedure somewhere while exiting. SET-SCREEN-MODIFIED gets reset."

TABLE A4
Program model—part 2

SYS23—Search for var definitions/use Comments indicating that a specific data structure is being sought for examination. For example, "I'm looking for an **Xvar** since it thinks that I went from 1 to len." This category does not include the actual data structure examination itself, this is covered by SYS5
SYS24—Search for begin/end of block Statements indicating that the beginning and/or end of a physical block is being sought after. For example, "This BEGIN goes with this END" or "I need to find the end of this while loop."
SYSCONF—Confirmed hypothesis Comments indicating that a previously stated program model hypothesis has been confirmed
SYSKNOW—Use of program model knowledge Statements expressing information in the program model, for instance program language syntax. References to specific language constructs or variables, statements and parameters. Any comments indicating that knowledge acquired sometime in the past either before or during the programming session is being used in reasoning. Include comments about previous sessions, for example, reading related sections of code. Also included are statements using knowledge already acquired in the on-going program session, for example, knowledge that is chunked and stored. "WITH OBJECT". This is PASCAL here that I am dealing with. It is not a standard WITH statement", is a good example.

Appendix B: Classification of model components—hypothesis codes

Only the action types that occurred in the corrective maintenance protocols are shown in Tables B1–B3. For example, SITH9 did not occur in the protocols and therefore does not appear in Table B2.

TABLE B1
Situation model hypothesis identification

SITH1—Variable function Hypotheses regarding the function of a variable at the algorithmic level. Example: "Yes. The name doesn't mean anything. It's only positional. You know what I'm saying how it's pushed onto the stack. Okay, let's see if I can find where message is used at. We've been through that, I don't think the message size has anything to do with the CONNECT STATE."
SITH2—Function/code block execution order/state Hypothesis about the execution order of functions or code blocks. Also includes the state of major components in the program. Example: "And it's probably WRITE_LOCKED_PIPE. And it's doing…Oh, okay. Writes the header, writes the data."
SITH3—Function/procedure function, call function Hypotheses about the functionality of a specific procedure or function and the functionality of a call to a function. Example: "Okay. Total bytes written…Okay, I think it's returning zero. But last time I thought that too. And that's not.."
SITH4—Effect of running program Hypotheses about what might occur if the program is run. Example: "No. I'm wondering if it's returning an address to an area in memory rather than directly in a register."

TABLE B1 (Continued)

SITH5—Cause of buggy behavior
Hypotheses regarding the cause of observed behavior considered incorrect. Example: "Okay, It suspect there's a problem in resolving the address in the CONNECT."

SITH6—Comparison of terms/acronyms/functionality
Hypotheses about whether terms or acronyms are similar to each other. Also includes comparisons of functions at an algorithmic level. Example: "So, this DO-ERASE-WRITE is very much like the DO-WRITE except for you erase the field before you do the write. Now that is based on an assumption here, I guess."

SITH7—Existence of function/algorithm/variable
Hypotheses regarding the existence of specific functionality, algorithms, and variables. Example: "Now what I didn't see, is any kind of call to an initialization routine and I don't know if I just missed that or there isn't one. I saw a little bit of setup they did. I thought in the call graph there was an initialization routine and I am going to go look at the call graph again. I thought there was something called … "

SITH8—Program function
Hypotheses about the function of a program at the algorithmic level. Example: "So I gather the reason they have all these routines is because they've got own malloc." **Only the action types that occurred in the corrective maintenance protocols are shown here. For example, SITH9 did not occur in the protocols and therefore does not appear in Table B2

TABLE B2
Domain model hypothesis identification

OPH1—domain: procedure function/concepts
Hypotheses about functionality of a procedure or function at the domain level, including domain concepts. Example: "But knowing a little bit about DOMAIN [operating system] now I know that's probably just an interface routine to DOMAIN system call."

OPH2—variable function/domain concepts
Hypotheses about functionality of a variable at the domain level, including domain concepts. Example: "Yeah, it looks like it may be constants. That uh refer to different attributes and things."

OPH3—Rules of discourse/expectations
Conventions in programming, similar to dialogue rules in conversation. Examples are coding standards, mnemonic naming, etc. Expectations of programmers. Example: "And I haven't been through this stuff at all, there is usually a list of file or routines right at the very beginning."

OPH6—Existence of specific functionality
Hypotheses regarding the existence of specific functionality in the program. Example: "Yeah. Okay. Assuming that it were, assuming that it's this name. Um, it would have to resolve this name with the link. During the link or something like that right?"

OPH7—Number/type/existence/location of libraries
Hypotheses about the number of libraries that exist, the type of libraries, whether they exist or their locations. Example: "One problem is that uh, because I'm not reality familiar with X-View I'm not too sure of what is part of X-View and what isn't. Although I'm pretty sure that XV stuff is going to be part of X-View."

OPH8—Program functions correctly
Hypotheses about whether the program is functioning correctly. Example: "So how did we get there? There's actually a chance that this is not a problem with the swapping code because I have been fiddling with … "

TABLE B2 (Continued)

OPH9—Permissions/environment set correctly/tool functionality
Hypotheses about whether permissions on files are set correctly or whether the environment is set up as expected and how tools, for example grep, behave. Example: "And the reason it probably doesn't like that, is that it has the path of the file where it originally was when it was built."

OPH11—Comparison of functionality at high level
Hypotheses about whether functionality between program fragments is similar. Example: "Yeah, so I'm going to add to the other task to also check …check file repair, because I think that works the same as the other one. And the rest of these may be the same."

OPH13—Number/type/location of file
Hypotheses about the number or type of files the subject is looking at or for. Also includes hypotheses about where certain files are located. Example: "There is a subdirectory under here I'm curious as to what it is. Who knows what that is, um, but it looks like just some text files."

OPH14—Available functionality
Hypotheses about functionality discovered during code comprehension. This is different from OPH6 where the programmer assumes some functionality exists. Example: "And there's some stuff, IFDEF for *MSDOS* which is interesting because I didn't know this ran on an MSDOS system, actually I guess I really don't know if it does or if they just have some support in there but haven't really tested it, or compiled it on there yet."

OPH16—Level & structure of code/scope
Hypotheses about the structure of the code, where in the call graph a certain procedure call is located. Example: "I am expecting it to be a second level procedure. So I am looking for the parent procedure and I don't seem to be finding it."

OPH17—Design decision/modifications
Hypotheses about which design decisions and modifications were made to the code. Example: "Ok, I think I understand the reason now, I said I assume they had a good reason, and I think they do. I think that if they wrote their own thing called malloc and linked it in they couldn't control the way in which STRING_DUP, the library routine called that, and what they are doing is they are checking the return code from malloc in only one place and then exiting the whole program if that failed and they only have to do that once. Whereas if they had their own malloc then they'd have to check the return code from STRING_DUP and everything else that called it in a lot of different places. Which would be more of pain. So, that's apparently the reason why there are doing what they are doing."

OPH18—Location/status/description/cause of error
Hypotheses about where an error is occurring, about the cause and description of the error. Example: "It's either the state is off by 1 by the time we get to here, but somehow that, . . I just don't understand that. And I don't know enough about ethernet itself to know. It seems like there's like some sort of hardware framing that must go on."

OPH19—Current location
Hypotheses regarding example: "(computer beeps) Looks like we're in another file."

TABLE B3
Program model hypothesis identification

SYSH1—variable function
Hypotheses about function of a variable at the code level. Example: "We're going to send a nil, now I'm concerned about what nil might be. Up here it was, when we did PUTREC, we sent the address of something. Here it looks like what were saying is well, point it at the zero location in memory. Whatever is there we're going to send two bytes of it, but we're going to say the length is zero"

SYSH2—Function/procedure function
Hypotheses about the function of a procedure or function at the code level. Example: "So when invoke PACKET_SEND_INTERNET so I'm going to try and ??? packets and here it is, I think. I have a feeling that things are just going to be passed in here. Packets entered in and called just Passed. Passed length and things like that."

SYSH4—Variable structure
Hypotheses regarding the code structure of a particular variable. Example: "Okay so VFORMAT DECODE2 um, decodes, uh, CONTROL_STRING is a format string defining the fields in the source string to be decoded. So we're going to decode this guy which, and the second thing is going to be length. I believe."

SYSH5—Location/type/existence of function call
Hypotheses about the location or type of a function called at a specific location in the code. Also includes hypotheses regarding the existence of a function call some place within the code. Example: "It looks like it's probably an internally called routine."

SYSH6—Statement execution order/state
Hypotheses regarding the sequence of statement execution or the state of the program, e.g. current rogram counter. Example: "and then if it was okay, um then what to do is jump around this BEGIN, this block starts with ELSE_BEGIN and um, it seems to me that we would still want to do the CALL. And so what I'm puzzled over is that if that was a, it looks like to me right here we start on the assumption that the first token was clear or disconnect."

SYSH7—Variable value/defaults
Hypotheses about the value a particular variable contain or default value of a specific variable in the program code. Example: "this looks like what is going to do is just load um, here's this X3 guy that up here in INIT_X3 we set up what I would guess are the default values."

SYSH8—(Non)Existence of construct (variable or code)
Hypotheses about the existence or non-existence of a variable procedure, function or some other code construct. Example: "So string table create. We'll assume that it's in here"

SYSH9—Variable/construct equivalency
Hypotheses about whether two variables are constructs are equivalent. Example: "CREATE RESOURCE TABLE. Well I can imagine that's going to look the same."

SYSH10—Syntax meaning
Hypotheses about the syntactical meaning of a specific code construct. Example: "There is a I guess a reserved word here that I am not completely familiar with."

SYSH13—Code block/procedure comparison
Hypotheses about whether blocks of code or procedures are structured similarly. Example: "Okay so DO-WRITE-DATA-STREAM-MODE called SET-SCREEN-MODIFIED. Based on the other two on the conclusion of the other two I am going to assume that it is very similar."

SYSH16—Code correctness, cause/location of error
Hypotheses about whether the code looking at is actually correct, the cause and location within the code of the error. Example: "I don't think it's in the syntax part. Because in syntax, I didn't seem to

TABLE B3 (Continued)

get any errors that didn't do the work itself. It does the d-link and the link. Because it disappeared, you tend to think the d-link worked and the link didn't work. The link is in this procedure called SCHEDULE-SCHED."

SYSH18—Location to add code/alternatives
Hypotheses regarding the best location to add specific code and hypotheses regarding alternative coding solutions. Example: "it's a rather simple fix. Actually it's just an IF THEN and I need to change the IF condition here."

SYSH20—Parameters/type definitions in procedure call
Hypotheses about example: "So we do get a length passed in at least at this level. But that's the length. (pause) And that's … so this is the requesting side. So RUN_FILE for HARD_LINK. That gets a LENGTH coming in. That's LENGTH of _LEAF_FORM I'd expect um, I'm looking for another thing. And I don't see it there which kind of … You got the LINK you want to add, create and you also have the TEXT of the LINK. And I only see one of those coming in. So I'm a little confused.