PROGRAM UNDERSTANDING BEHAVIOR DURING DEBUGGING OF LARGE SCALE SOFTWARE

Anneliese von Mayrhauser and A. Marie Vans
Colorado State University
601 Howes Lane, room 239
Fort Collins, CO 80523 USA
[avm, vans] @cs.colostate.edu

KEYWORDS: Program Comprehension, Software Maintenance, Debugging.

ABSTRACT

This paper reports on a software understanding experiment during 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, and the level of abstraction at which they were working. The results of the observation are also interpreted in terms of the information needs of these software engineers during the debugging task.

1. INTRODUCTION

Program understanding is a central activity in a variety of maintenance tasks. During corrective maintenance, the software engineer needs to understand the software enough to analyze a problem, locate the bug, and determine how it should best be fixed without breaking anything. For larger software products, understanding will 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, the observations considered both situations. The questions we tried to answer were:

1. What kinds of actions do programmers perform when debugging code?
2. Do programmers follow the Integrated Comprehension model of (von Mayrhauser & Vans, 1995a)? Do they switch between its three model components? Is there a preference for a particular model component?
3. Is it possible to identify a specific comprehension process that is common to the subjects and thus indicative of debugging tasks?
4. Are there certain types of information programmers tend to look for during corrective maintenance?

We started our investigation with the premise that in industry, large-scale programs are a prevalent focus for software comprehension and corrective maintenance activities. von Mayrhauser and Vans (von Mayrhauser & Vans, 1995a, 1993a, 1993b) describe an Integrated Comprehension Model that characterizes large scale code comprehension activities as a combination of three comprehension processes at the program, situation and top-down (application domain) levels. Section 2 explains this Integrated Comprehension Model and provides the background for an observational study of software maintenance engineers working on corrective maintenance tasks.

Section 3 describes the design of the study. 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 regards 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. INTEGRATED COMPREHENSION MODEL (VON MAYRHAUSER & VANS, 1993A)

Existing code cognition models agree that comprehension occurs either top-down, bottom-up, or using a combination of both. Studies with large scale code (von Mayrhauser & Vans, 1993a, 1995a) indicate that code comprehension include both top-down and bottom up activities. Soloway and Ehrlich's model (Soloway, Adelson, & Ehrlich, 1988a) is the foundation for the top-down component (the domain model) while Pennington's model (Pennington, 1987a, 1987b) motivated the program and situation models. The Integrated Comprehension Model contains four major components: (1) *Program Model*, (2) *Situation Model*, (3) *Top-Down Model* (or domain model), and (4) *Knowledge Base*. The fourth is necessary for construction of the other three models. Program, situation, and top-down (or domain) model building form the three processes that direct the understanding of code. As Figure 1 demonstrates, any of these comprehension processes may be activated from any of the others. Beacons, goals, hypotheses, and strategies determine the dynamics of the cognitive tasks and the switches between the model components. Each component carries an internal representation (mental model) of the program being understood. This representation differs in level of abstraction for each model component. Each component also includes strategies to construct this internal representation. The knowledge base supplies the processes with information related to the comprehension task. It also stores any new and inferred knowledge.

The *Top-Down* model of program understanding is typically invoked during the comprehension process if the code or type of code is familiar. The top-down model or domain model represents knowledge schemas about the application domain. For example, a domain model of an Operating System (OS) would contain knowledge about the components of an OS (memory management, process management, OS structure, etc.) and how they interact with each other. This knowledge often takes the form of specialized schemas including design rationalization (e.g., the pros and cons of First-Come-First-Serve versus Round Robin scheduling.) Obviously, a new OS will be easier to understand for a maintenance engineer 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 high paging rates requires understanding how process scheduling and paging algorithms are implemented and whether the system limits the number of pages allocated to processes).

When code to be understood is completely new to the programmer, Pennington (Pennington, 1987a, 1987b) found that the first mental representation programmers build is a control flow abstraction of the program called the *program model*. For example, operating system code may be understood by determining the control flow between modules. Then a single module may be selected for content analysis, e.g. a scheduling function. This may use an implementation of a doubly-linked list. The code representation is part of the program model. The abstraction of the scheduling queue as a doubly-linked list is part of the situation model representation.

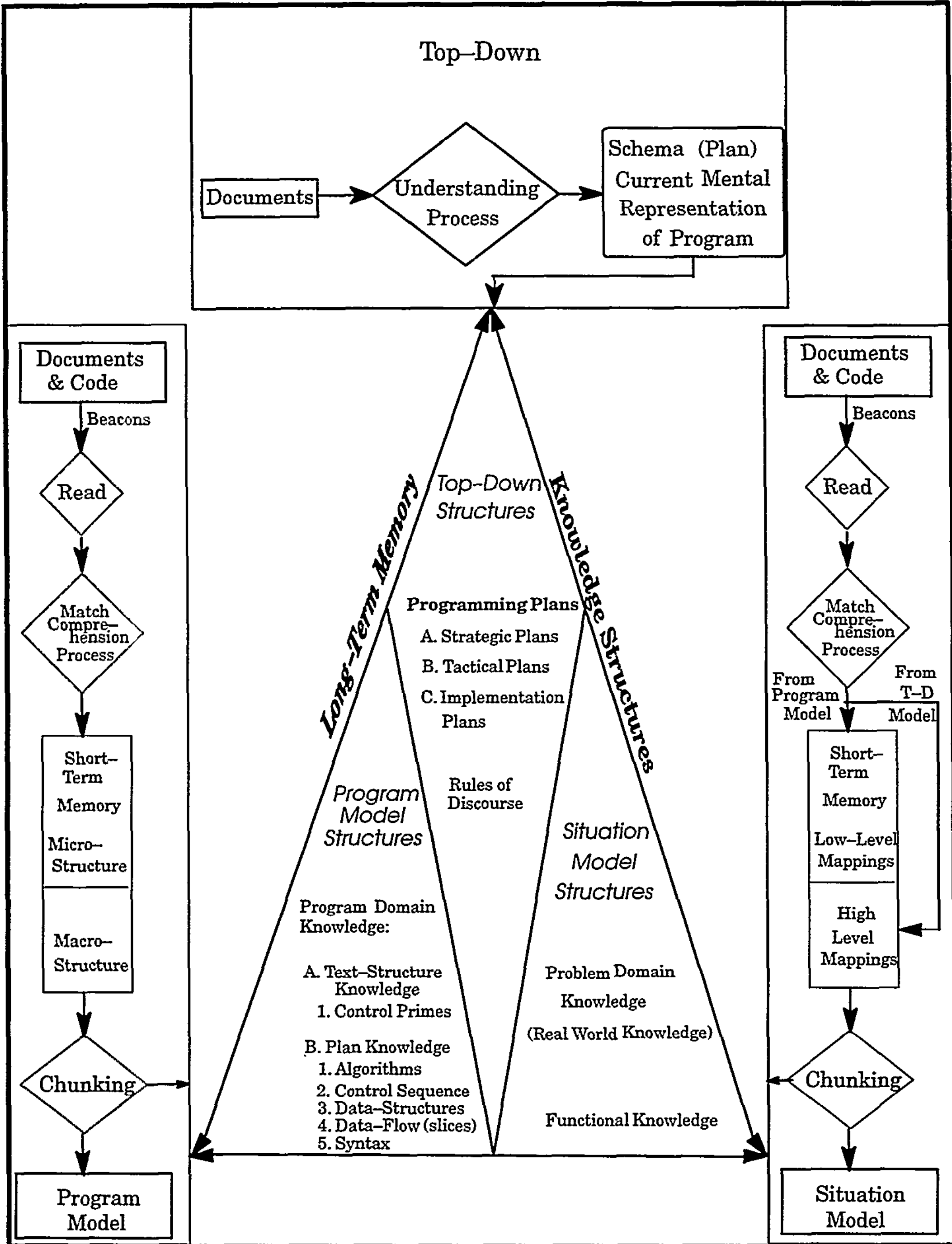

Figure 1: Comprehension Model (von Mayrhauser & Vans, 1993a, 1995a)

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 also assumes that maintenance engineers unfamiliar with the domain start by building a program model. However, to assume that a full program model is built before abstracting to the situation or domain level would create cognitive overload for professionals working on software products with over 40,000 lines of code. Rather, what we expect is abstraction of program model information at the situation and domain level as it helps the programmer remember how the program works and what it does.

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.[1] If, for example, a beacon leads to the hypothesis that a sort is performed, the switch is to the top-down model. The programmer then generates sub-goals to support the hypothesis and searches the code for clues to support these sub-goals. If, during the search, a section of unrecognized code is found, the programmer jumps back to building the program model. Figure 1 illustrates the relationships between the three sub-models and the related knowledge.

The definition of the integrated model allows a refinement in terms of tasks and task sequences for each of the three comprehension processes. Comprehension is further guided by the systematic bottom-up, the opportunistic top-down, or a mixed systematic/opportunistic strategy.

A systematic approach is one in which the programmer applies a systematic order to understanding code completely, for example code comprehension line by line. An opportunistic approach involves studying code in an as-needed fashion. The distinction between *systematic* and *opportunistic* (or *as-needed*) is important because Littman et.al. (Littman, Pinto, Letovsky, & Soloway, 1986) found that programmers who use a systematic approach to comprehension are more successful at modifying code (once they understand it) than programmers who take the opportunistic approach. Although the systematic strategy seems *better* or *safer*, it is unrealistic for large programs. A disadvantage to the opportunistic approach is that understanding is incomplete and code modifications based on this understanding may be error prone (Littman et. al, 1986).

Various aspects of this model have been confirmed in prior studies. von Mayrhauser and Vans (von Mayrhauser & Vans, 1993b) 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 (von Mayrhauser & Vans, 1993a) extended these results to include a debugging task. They also analyzed for detailed action types, in addition to actions by component model. Observations are interpreted in terms of possible tool capabilities. von Mayrhauser and Vans (von Mayrhauser & Vans, 1994a, 1996b, 1996c) investigated cognition processes to confirm through observations the processes stipulated in the model. von Mayrhauser and Vans (von Mayrhauser & Vans, 1994a, 1996c) report on the comprehension process of one subject who had used a systematic understanding strategy where actions could be aggregated into episodes, episodes into aggregate processes, and those in turn into a session level process. The analysis found 7 types of episode level processes, 3 aggregate level processes, and one session level process. The episode level processes include switches between model levels and thus are not pure bottom-up understanding processes. By contrast, von

[1] For example, a beacon may be the characteristic pattern of value switches indicating a common task (such as sorting), or it may be the name of the function (such as QSORT).

Mayrhauser and Vans (von Mayrhauser & Vans, 1996b) report on the comprehension process of an engineer who was porting software and employed a fundamentally different comprehension process related to an opportunistic strategy. This process was driven, and could be structured around a hierarchy of goals, hypotheses, and actions. These results support the integrated model, the switching behavior between model components, and the role of hypotheses in an opportunistic understanding process.

3. STUDY DESIGN

The experiment was designed as an observational field study of professional maintenance programmers working on software maintenance. Software consisted of at least 40,000 lines of code. Each observation involved a *programming session* in which the participants were asked to think aloud while working on corrective maintenance. We audio and/or video taped this as a thinking aloud report. Sessions were typically two 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 distinguishes between programming knowledge (language and platform skills) and application domain knowledge. This is the same classification used in earlier studies (von Mayrhauser & Vans, 1995a, 1996c). Shaft and Vessey (Shaft & 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. We also consider the amount of accumulated knowledge, as it 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. The *type of maintenance task* was corrective maintenance. Table1 describes the attributes of the four subjects and the software they worked on.

Table 1: Subject Classification

Subject	Task	Expertise	Accumulated Knowledge
C1	Fix reported bug	Language expert (Pascal) novice in application domain (communication protocols)	Some knowledge: File structure, call graph, requirements/design
C2	Understand bug	Language novice (expert in Microsoft Windows and C, but not XT Intrinsics); domain expert (software project management software)	None
C3	Fix reported bug	Language novice (expert in C, but not Pascal); domain expert (OS kernel)	Little (file structure, call graph)
C4	Track down bug	Language novice (expert in C, but not Pascal); domain expert (client-server, OS)	Significant:(prior maintenance tasks on same software)

The code the participants tried to understand and the specific assignment are representative of this type 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, 1996b) 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, 1993a, 1993b, 1995a, 1996a,1996c). Our objectives were to analyze the observations for characteristic activities and behavior. The analysis also identified information needs. We used protocol analysis for this purpose.

3.1. Protocol Analysis

The same protocol analysis methods employed here were used in prior analyses (von Mayrhauser & Vans, 1993a, 1993b, 1995a, 1996a, 1996c). Think-aloud reports of subjects working on tasks are transcribed and classified using categories decided on prior to the actual analysis. For example, we expect to find maintenance engineers generating hypotheses and reading code during maintenance. Each statement in the transcript is *encoded* as one of the a priori categories. Thinking aloud must occur concurrently with the task for the data to be accurate. The analysis proceeds from identifying single actions of various types to determining action sequences. The analysis parallels experimental work related to the Integrated Code Comprehension Model. This Integrated Model was developed to explain how programmers understand large-scale code in industrial settings (von Mayrhauser & Vans, 1994a, 1995b). This model is the framework for discovering the role of actions, hypotheses, and comprehension behavior in large-scale code understanding. The integrated model consists of four components: 1) Top-Down model, based on Soloway and Ehrlich's model (Soloway & Ehrlich, 1994); 2) Situation Model, based on Pennington's situation model (Pennington, 1987a); 3) Program Model, also based on Pennington's model; and 4) a knowledge base which is used to store and retrieve information used by the three component models. A detailed description of the model can be found in (von Mayrhauser & Vans, 1994a, 1995b).

Actions

The first analysis on the protocols involved *enumeration* of *action types* as they relate to the integrated cognition model of (von Mayrhauser & Vans, 1995b). Action types classify programmer activities, both implicit and explicit, during a specific maintenance task. Examples of action types are "Generating hypotheses about program behavior" or "mental simulation of program statement execution". We began with a list of actions we expected to find based on (Vessey, 1985) and searched for them in the transcripts of the protocols. We also analyzed for possible new action types.

Segmentation and 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 (top-down), situation, or program model that can be thought of in terms of different levels

of abstraction in the mental model. *Information Needs* are information and knowledge items that support successful completion of maintenance tasks. For prior results with different subjects see (von Mayrhauser & Vans, 1993a, 1993b).

Protocol analysis is an iterative process. A first pass analysis results in a high-level classification of programmer actions as either program, situation, or top-down model components of the Integrated Model. This is necessary because similar actions appear in different component processes. For example, hypotheses may be generated while constructing any of the three component models. Once actions are associated with a particular model component, the next pass identifies action types of a specific maintenance task. Once the action types are identified, the transcripts are re-analyzed and encoded using these types as tags on the programmer utterances.[2] Information Needs are determined from protocols directly (see Table 3) or through inference.

Table 2: Example Protocol Analysis -- Action Types

Tag	Action Type	Example Protocol
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

Tag	Information Need Classified As:	Example Protocol
I9	**List of Browsed Locations** OP2, OP7, SITKNOW, OPKNOW, SYS10	"Because if class SPOOL-INTO, yeah. So I was at the right place. A long time ago."
I61	**General Classification of Related Functions** OP3, OP4, OP14, OP15, OP20, SIT4, SYS8, SYS11	"Yeah this is the one. DbFile Insert. Insert dbfile. Get the dbfile in. Curse the author of this one. Try to insert dbfile. So...it sounds a lot like the other one."

An information need may not be directly stated but the programmer could obviously profit from it if he knew it existed. For example in Table 3, the protocol segment associated with the *General Classification of Related Functions* information need demonstrates that if he was sure the particular function he was examining was related to a function he had seen before, his understanding of the current function might have occurred more quickly. He spent a great deal of time examining several documents for this very information.

Tables 2 and 3 contains example protocols to show action type classification and information needs identification. Column one of tables 2 and 3 provide the tag used in action type and information need classification.

Process Analysis

For actions, process analysis determined the nature of actions over time graphically. Each action has already been classified by level of abstraction (Program, Situation, or Domain level). They are

[2] Utterances are verbalizations of programmers during programming sessions and captured in the transcripts

now plotted as a function of "time" (in terms of numbers of actions). These 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.

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 give an indication of relative frequency of these actions for each subject.

Table 4: Action-Types by Model – Totals & Frequencies

Subject Code	Top-Down Model	Program & Situation Model	Situation Model	Program Model	Total Actions
C1 *Corrective*	46 13%	303 87%	160 46%	143 41%	349
C2 *Corrective*	119 33%	240 67%	91 25%	149 42%	359
C3 *Corrective*	119 37%	202 63%	42 12%	160 51%	321
C4 *Corrective*	171 48%	186 52%	101 28%	85 24%	357
Total Corrective	455 33%	931 67%	394 28%	537 39%	1386

The table contains a column for combined Program and Situation model references, roughly corresponding to Pennington's (Pennington, 1987a) comprehension model. We wanted to identify patterns based on differences between Pennington's bottom up model and the top-down model.

Both C2 and C3 had a similar distribution of top-down and combined program and situation model actions. Each had twice as many references to the combined program and situation models as to the top-down 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 (Pennington, 1987b) results that programmers tend to build a program model first when they are unfamiliar with the code.

C1's model preferences were more dramatic. 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 13% of his model references were in the domain model. We believe that lack of domain experience drives the behavior we see. 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 into the domain model.

The last subject, C4, had both domain experience and significant experience with the code. He had an almost equal distribution of top-down 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 top-down model.

Tables 5, 6, and 7 show 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 for individual subjects. The last column contains the total number of each action for all subjects.

Table 5: Action Types -- Top-Down Model

Code	Action Type	C1	C2	C3	C4	TTl.
OP1	Gain high-level Program overview	6	16	0	5	27
OP2	Determine next prgm. Segment to examine	7	3	23	12	45
OP3	Generate/revise hypothesis re: functionality	9	18	25	24	76
OP4	Determine relevance of prgm segment	1	8	1	8	18
OP5	Determine if prgm seg needs detail understand	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	1	0	1	0	2
OPKNOW	Use of Top-down knowledge	6	24	38	39	107
Total	**Top-Down Model Actions**	49	120	121	171	461

Table 6: Action Types -- Situation Model

Code	Action Type	C1	C2	C3	C4	TTl.
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
SIT4	Chunk & store	41	29	13	24	107
SIT5	Determine relevance of situation knowledge	5	4	2	4	15
SIT6	Determine next info to gain	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	1	1	0	1	3
SITKNOW	Use of Situation model knowledge	29	17	18	28	92
Total	**Situation Model Actions**	162	92	42	103	399

Table 7 : Action Types -- Program Model

Code	Action Type	C1	C2	C3	C4	TTl.
SYS1	Read intro code comments/related docs	4	15	7	1	27
SYS2	Determine next prg segmt to examine	15	7	3	1	26
SYS3	Examine next module in sequence	21	29	43	9	102
SYS4	Examine next module in cntrl-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 questions	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 variable definitions/uses	0	1	3	0	4
SYS24	Search for block begin/end	2	0	0	0	2
SYSCONF	Confirmed hypothesis	0	0	4	1	5
SYSKNOW	Use of Program model knowledge	24	5	21	11	61
Total	Program Model Actions	145	151	166	89	551

For top-down model building, using domain knowledge is clearly the most frequent action (count=107). Domain knowledge was used 23% of the time during top-down model construction. This supports our hypothesis that when programmers have domain knowledge they use it frequently for building a top-down model. Of the four subjects, three were domain experts (C2, C3, and C4) and they had 94% of the references to top-down knowledge. The second most

important action type for top-down model building is generating or revising hypotheses (*OP3*, count=76). Using hypotheses for building a top-down model is a feature of Brooks' theory of program comprehension (Brooks, 1983).

For situation model building, chunking and storing acquired information (*SIT4*, count=107) was most important. This confirms 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 comfort at the lower levels and unfamiliarity with the domain. Similar to the domain level, both use of situation model knowledge (count=92) and generating hypotheses (*SIT7*, count=64) are important activities. This confirms Pennington's model, but also supports hypotheses as being major drivers of comprehension. These are the second and third most frequent actions at the situation model level.

At the program model level, chunking and storing knowledge (*SYS7*, count=115) is the most important action. Again, this confirms Pennington's theory that programmers build higher levels of abstraction from low level information. Examining code in sequence (*SYS3*, count=102) and generating hypotheses (*SYS8*, count=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 (count=61).

The use of knowledge and generating hypotheses are important activities for programmers. In all three model components of the Integrated Model, these action 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, Pinto, Letovsky, & Soloway, 1986).

The types and distribution of actions between the models tells us something about the level of abstraction in 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. We will examine this next.

4.2. Processes

Switches occur between all three models (top-down, program, and situation models). Table 8 summarizes switches between models during the corrective maintenance task. 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 functionality. (A switch from situation to program model.)

Table 8: Action Switches - Absolute & Frequency (Total switches = 562)

From Model	To Top-Down Model	To Situation Model	To Program Model
Top-Down Model	N/A	67 12%	93 17%
Situation Model	86 15%	N/A	112 20%
Program Model	75 13%	129 23%	N/A

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, 3, 4, and 5 illustrate how these switches happen over time. These graphs are modeled after those found in Pennington et.al (Pennington, Lee, & Rehder, 1995) and Lee and Pennington (Lee & 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, longer numbers of actions between switches are indicated by a shallow incline or decline in the switch-line.

C1: Language Expert, Some Accumulated Knowledge

Table 9: C1: Action Switches Frequencies Between Models (Total Switches = 118)

From Model	To Top-Down Model	To Situation Model	To Program Model
Top-Down Model	N/A	12%	12%
Situation Model	15%	N/A	25%
Program Model	9%	27%	N/A

Table 9 shows the frequency of switches between each of the three models. C1 had a total of 118 switches between models. Switches between the situation and program model are more frequent than switches between either the top-down and program models or the top-down and situation models. We hypothesize this is due to the subject's lack of domain knowledge.

Figure 2: C1: Fix Reported Bug -- Action Sequence

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 top-down model. The graph also shows that he spent more action time either in the program or situation models since the jumps into the top-down 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: Domain Expert, No Accumulated Knowledge

Table 10: C2: Action Switches Frequencies Between Models (Total Switches = 192)

From Model	To Top-Down Model	To Situation Model	To Program Model
Top-Down Model	N/A	12%	16%
Situation Model	14%	N/A	21%
Program Model	14%	23%	N/A

Table 2 shows how often C2 switched between models. This programmer switched a total of 192 times during the programming session. C2 switched slightly more often between the top-down and program models than C1. However, switches between the program and situation model occurred at the highest frequency.

Figure 3: C2: Understand Bug -- Action Sequence

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 spent significant amounts of time in the domain model before switching to another level, he spent more action time in the top-down model than did C1. Because he was unfamiliar with 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 top-down levels. The majority of the switches into the situation model occurred from the program model.

C3: Domain Expert, Little Accumulated Knowledge

Table 11: C3: Action Switches Frequencies Between Models (Total Switches = 127)

From Model	To Top-Down Model	To Situation Model	To Program Model
Top-Down Model	N/A	5%	27%
Situation Model	12%	N/A	15%
Program Model	19%	22%	N/A

Similar to C2, C3 switched frequently between the program and top-down models, however, he did not use the situation model as a bridge from the top-down to the program model as often as C2. Instead, he preferred to work at a particular model level, switching less frequently than C2. (C2 had a total of 192 switches and C3 had a total of 127 switches.) This is probably due to individual differences in understanding strategy.

Figure 4: C3: Fix Reported Bug -- Action Sequence

An important result here is that domain expertise may have an effect on this programmer's ability to switch directly between the top-down and program model levels. Accumulated knowledge may also be playing a part in that lack of experience may be driving this programmer's need to build a mental representation of the code in the most efficient manner. Having a top-down mental representation allows the programmer to quickly isolate code that needs correction.

C4: Domain Expert, Significant Accumulated Knowledge

Table 12: C4: Action Switches Frequencies Between Models (Total Switches = 125)

From Model	To Top-Down Model	To Situation Model	To Program Model
Top-Down Model	N/A	19%	11%
Situation Model	22%	N/A	18%
Program Model	10%	20%	N/A

C4 had a total of 125 switches between models. Switching is distributed pretty evenly among all three models except for direct switches between the top-down 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 we saw could be indicative of trying to fill in the holes at each level.

Figure 5: C4: Track Down Bug -- Action Sequence

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 were spent between the top-down and situation model levels. The next 50 between the program and situation model levels. Actions 100 to 150 are dominated by switches between the top-down and program model levels. Then he repeats this pattern for a short time before concentrating on the top-down 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.

Comparing the four subjects, we can make several conjectures about 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 will allow 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 will also work at lower levels of abstraction, but will be able to more effectively 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 top-down 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. The hypothesis is that they already have a good mental representation at all three levels and use switches as a means for completing the full model.

4.3. Information Needs

Information Needs are developed by analyzing each action type for the kind of information it needed or searched for. The results of this analysis are summarized in Tables 13 and 14. The information needs tables contain eight columns. The first provides a code for the information

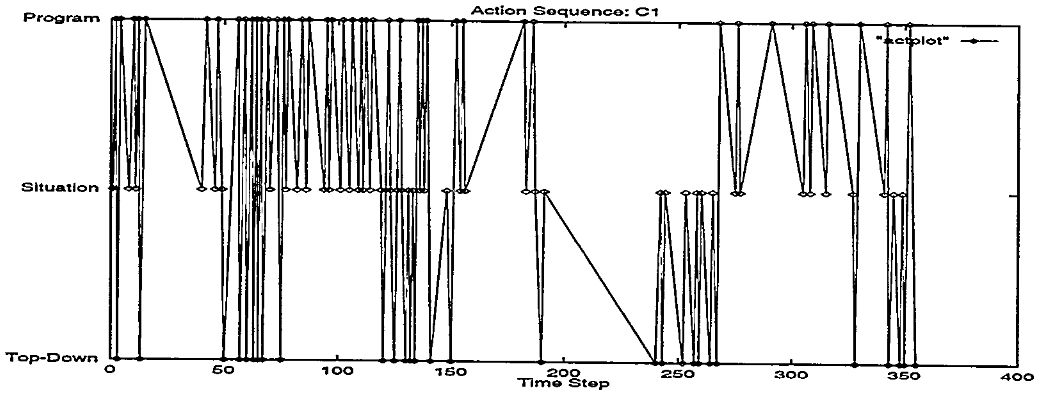

Figure 2: C1: Fix Reported Bug – Action Sequence

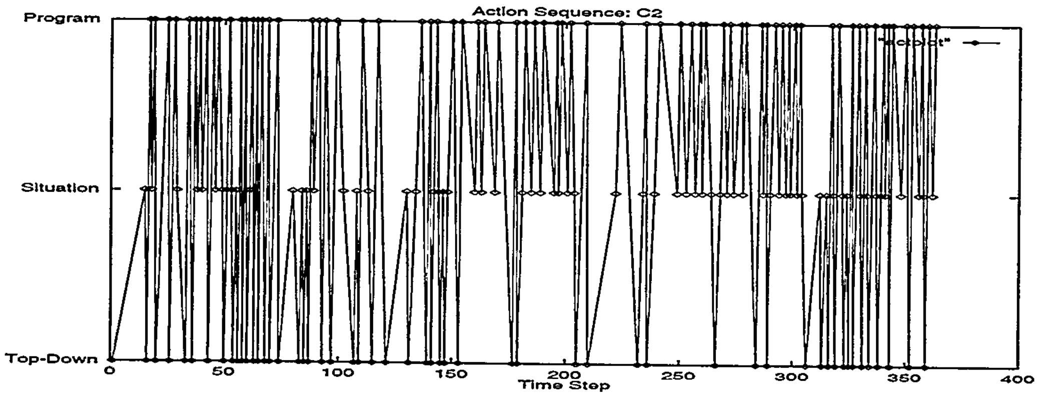

Figure 3: C2: Understand Bug – Action Sequence

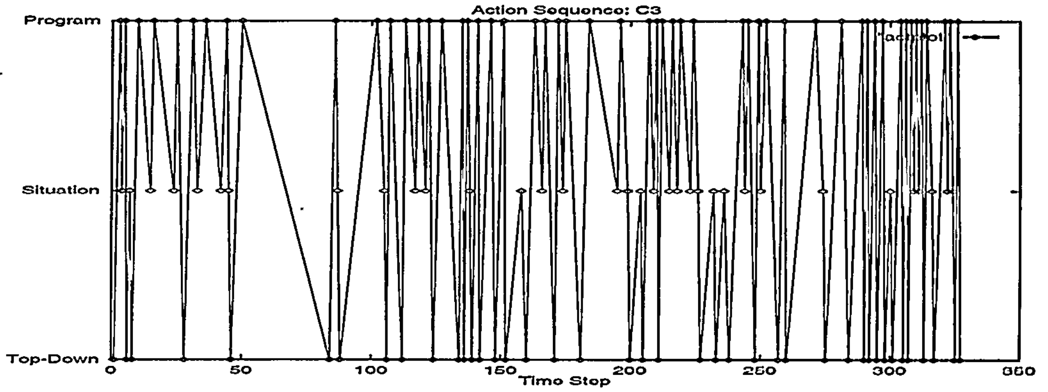

Figure 4: C3: Fix Reported Bug – Action Sequence

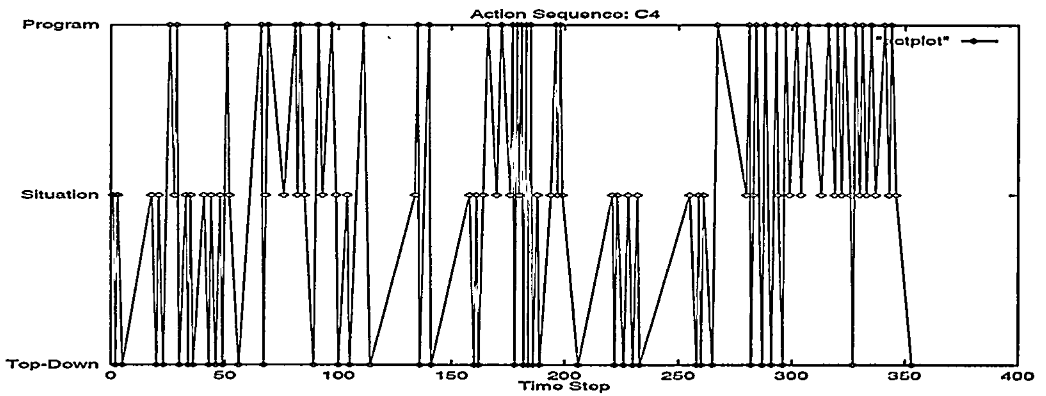

Figure 5: C4: Track Down Bug – Action Sequence

need. The second describes the information for which the software engineer is looking. The third lists the action type codes (associated with a specific model) for which the information is needed. The fourth lists cumulatively how often the subjects had a need for this information. The last four columns provide detail results for **C1**, **C2**, **C3**, and **C4** individually. These results help to illustrate the type of work a software engineer was doing and to prioritize solutions that would facilitate this work.

Table 13: Information Needs

Code	Information Need	Action Codes	Sub. TTI	C1	C2	C3	C4
I7	Domain concept descriptions	OP1,OP3,OP6, OP17,OP20, OPKNOW,SIT1, SIT2,SIT4,SIT6, SIT7,SIT8,SYS1, SYS8,SYS10	41	14	18	2	7
I61	Connected domain-program-situation model knowledge	OP2,OP3,OP6, OP20,OPKNOW, SIT1,SIT2,SIT4, SIT6,SIT7,SIT8, SITKNOW,SYS1, SYS2,SYS3,SYS4, SYS7,SYS8,SYS10 , SYS11, SYS14	33	5	12	10	6
I4	Location and uses of identifiers	OP2,OP6,OP20, OPKNOW,SIT5, SIT6,SIT7,SIT8, SITKNOW,SYS2, SYS4,SYS7,SYS8, SYS10,SYS11, SYS23,SYSKNOW	26	3	8	8	7
I9	List of browsed locations	OP2,OP3,OP15, OPKNOW,SIT2, SIT4,SITKNOW, SYS7,SYS10, SYS11,SYS12, SYS13,SYSKNOW	17	2	7	8	0
I2	List of routines that call a specific routine	OP2,OP3,OP6, SYS2,SYS7,SYS9, SYS10,SYS11, SYS12	14	7	1	6	0
I14	Call Graph Display	OP2,OP3,OP6, OP15,OP20, OPKNOW,SYS9, SYS13	12	3	4	5	0
I73	Bug behavior isolated	OP3,OPKNOW, SIT4,SIT7,SIT8, SITKNOW,SYS3, SYS7,SYS8,SYS19 ,SYSKNOW	11	1	0	5	5
I43	General classification of routines/functions	OP2,OP3,OP14, OPKNOW,SIT1, SIT7,SIT12,SYS4, SYS8,SYS10	11	4	6	1	0
I5	Format of data structure plus description of what field is used for in program and	OPKNOW, SITKNOW,SYS3,	8	4	1	3	0

	application domain, expected field values and definitions	SYS5,SYS7,SYS8, SYSKNOW					
I22	History of past modifications	OP3,OP17, OPKNOW,SYS8	7	3	2	0	2
I24	List of executed statements and procedure calls, variable values	OP3,OP20, OPKNOW,SIT2, SYS7,SYS8	7	0	0	0	7
I66	Expected program state, e.g. expected variable values when procedure is called	SIT2,SIT7,SIT8, SITKNOW,SYS10	6	2	0	0	4
I72	Good direction to follow given what is already known, possible program segments to examine	OP3,OP18, OPKNOW,SIT1, SYS2,SYS10	6	1	3	2	0
I27	Directory layout/organizations: include files, main file, support files, library files. File structure	OP6,OP20	5	1	1	0	3
I36	Sequence of locations where ID is used	OP3,OP6, OPKNOW,SYS10, SYSKNOW	5	0	0	5	0
I38	Nesting level of a particular procedure	OP2,OP3,OP6,OP8, SYS3	5	5	0	0	0
I44	List of routines that do most of the domain-type work	OP4,OPKNOW, SITKNOW, SYSKNOW	5	0	4	1	0
I78	Dynamic function return values	SIT7,SYS8,SYS11, SYS12,SYSKNOW	5	0	0	0	5
I13	Conditions under which a branch is taken or not. Include variable values	SYS3,SYS7,SYS8	4	4	0	0	0
I16	Naming conventions separated by system or library objects that use them. Rules used for naming new identifiers	OP20,OPKNOW	4	0	4	0	0
I58	High-level description of how code is laid out	OP1,SIT1,SYS8, SYS10	4	1	3	0	0
I6	Location and description of library routines and system calls	OP3,OP6, OPKNOW,SIT7	3	0	2	1	0
I12	Environment, global, local scope	OP7,OP20, SITKNOW	3	0	1	0	2
I19	Acronym definitions	SIT2,SIT7	3	3	0	0	0
I20	Documentation list and location	OP3,OP17, OPKNOW	3	0	2	1	0
I39	Main program location	OPKNOW,SYS11, SYSKNOW	3	0	3	0	0
I62	Predefined (constant) variables and values	OP2,OP3,SYS2, SYS8	3	1	1	1	0
I68	List of issues/decisions considered during design	OP3,OP15,SYS12	3	1	1	0	1
I74	Good description of the bug and why and how other fixes were done	OP3,OP20, SYSKNOW	3	1	0	0	2
I76	Available tools to aid in understanding	OP17,OP20	3	0	0	0	3
I1	Variable definitions including why necessary and how used, default values and expected values	SIT2,SYS7	2	0	0	0	2
I3	Highlighted begins/ends of control blocks	SYS10,SYS24	2	2	0	0	0
I8	List of system calls based on specific naming convention	OP3,OP6	2	0	2	0	0
I11	Language conventions that are different from standard conventions. Lang	SYS8	2	2	0	0	0

	enhancements						
I21	Organized functions into categories in which functions are related	OP6,OP15,SYS10	2	0	1	1	0
I30	Where variable is toggled, when and why, where passed to and why	OP3,SIT7,SYS10	2	2	0	0	0
I37	Language definitions, e.g. reserved words, instruction defs, for C, Pascal,etc.	SYS2,SYS3,SYS8, SYS11	2	0	0	1	1
I41	Call graph with extraneous information not relevant elided	OP3,OP15	2	0	0	2	0
I49	File name of current file	SITKNOW, SYSKNOW	2	0	1	1	0
I55	Domain simulation	OP6,OP15	2	0	1	0	1
I63	All definitions and used of a variable prioritized..so more important uses show up first	SYS7,SYS23	2	0	0	2	0
I10	Function call count	SIT4,SYS3,SYS7	1	0	0	1	0
I17	Location of desired code segment	SYS12	1	0	1	0	0
I26	All include file definitions and uses	OP3	1	0	1	0	0
I28	List of all routines with initialization code	SIT4	1	0	1	0	0
I32	If common objects are not used in traditional way, e.g. nil or null	SYS8	1	0	0	1	0
I47	List of identifiers and domain concepts that are important	OP4	1	0	1	0	0
I59	How a variable is passed into a procedure, e.g. by value, address, etc	SYS10	1	0	0	1	0
I65	Assembly language code segment number (Machine Code)	SYS3,SYS7	1	0	0	0	1
I77	Data-flow trace	SIT2	1	0	0	0	1
	TOTALS		294	72	93	69	60

Overall, the most important types of information needed during corrective maintenance are domain concept descriptions (*I7*) and connected domain-program-situation model knowledge (*I61*), 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. It is not surprising that these two types of information are needed significantly 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 example occurs when a programmer skims data she does not fully understand. She needs to revisit the information when other data triggers a recall and subsequent comprehension of the previously skimmed data.

We are also interested in looking at the effect of expertise and amount of accumulated knowledge during corrective maintenance.

C1 needed domain concept descriptions most often (*I7*,count=14). The second most frequent need was a list of routines that call a specific routine (*I2*,count=7). The third most frequent was connected information from each of the models into the other models (*I61*, count=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 usually very important during initial comprehension. It becomes less important once the programmer has worked with code for a while.

For **C2**, domain concept descriptions (*I7*,count=18), connected model information (*I61*, count=12), and location and uses of identifiers (*I4*,count=8) were the top three most frequent 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 greatest amount of information. He had 32% of all the 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 corrective task than domain expertise or accumulated knowledge.

C3 needed connected model information (*I61*,count=10), location and uses of identifiers (*I4*,count=8), a list of recently browsed code locations (*I9*,count=8), and a list of routines that call a specific routine (*I2*,count=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 code.

Domain concept descriptions (*I7*,count=7), location and uses of identifiers (*I4*,count=7), control-flow graph (*I68*,count=7), and connected model information (*I61*,count=6) are needed most often by **C4**. This subject had the smallest number of information needs of all the 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 in the types of information he required.

In general, the need for understanding domain concepts and building the connections between the three model components can be driven by different goals. For programmers who have domain experience the need for this type of information may be due to the fact that they want 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 they can understand functionality of the code. We also found that the subject with the most experience, C4, had less information needs than the other three. While it was not significantly less, it could be an indication that as expertise and experience with the software increases, the need for information decreases.

5. CONCLUSIONS

Corrective maintenance is a frequent activity during software evolution. We observed four experienced professional programmers while they were debugging software and analyzed their behavior. The goal was to answer several questions about how programmer 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 a bridge between the program and top-down models. Domain expertise and significant experience with the software allows connections between all three levels of abstraction to be easily made.

3. Information Needs.

Domain concepts and connected program, situation, and top-down 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.

We consider these conclusions to be 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.

6. REFERENCES

Brooks, Ruven (1983), Towards a theory of the comprehension of computer programs. In *International Journal of Man-Machine Studies*, Vol.18, (pp. 543-554).

Lee, A. and Pennington, N. (1994), The Effects of Paradigm on Cognitive Activities in Design In *International Journal of Man-Machine Studies,* Vol. 40, (pp.577-601).

Letovsky, Stanley (1986), Cognitive Processes in Program Comprehension, In . E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*, (pp. 58 - 79). Norwood, NJ: Ablex.

Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. (1986), Mental Models and Software Maintenance, In . E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*, (pp. 80 - 98). Norwood, NJ: Ablex.

Pennington, N., Lee, A. Y., and Rehder, B. (1995). Cognitive Activities and Levels of Abstraction in Procedural and Object-Oriented Design, In: ***Human-Computer Interaction***, Vol. 10, (pp.171-226).

Pennington, N., (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs, In: ***Cognitive Psychology***, Vol. 19. (pp.295-341).

Pennington N., (1987). Comprehension Strategies in Programming. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), ***Empirical Studies of Programmers: Second Workshop*** (pp. 100 - 112). Norwood, NJ: Ablex.

Shaft, T. M. and Vessey; I., (1996). Computer Program Comprehension Processes: the Effect of Application Domain Knowledge, In: ***Empirical Studies of Programmers: 6th Workshop***, (pp. 277 - 278). Norwood, NJ: Ablex.

Soloway E. and Ehrlich, K., (1984). Empirical Studies of Programming Knowledge, In: ***IEEE Transactions on Software Engineering***, Vol. SE-10, No. 5, (pp. 595-609).

Soloway, E., Adelson, B., and Ehrlich, K., (1988). Knowledge and Processes in the Comprehension of Computer Programs, In M. Chi, R. Glaser, and M.Farr (Eds.), ***The Nature of Expertise***, (pp. 129-152). Lawrence Erlbaum Associates.

Vessey, I., (1985). Expertise in debugging computer programs: A process analysis, In ***International Journal of Man-Machine Studies***, Vol. 23, (pp.459-494).

von Mayrhauser, A. and Vans, A., (1993a). From Program Comprehension to Tool Requirements for an Industrial Environment, In ***Proceedings of the 2nd Workshop on Program Comprehension,*** Capri, Italy, (pp. 78 -86).

von Mayrhauser, A. and Vans, A., (1993b). From Code Understanding Needs to Reverse Engineering Tool Capabilities. In ***Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE93)***, Singapore, (pp. 230 - 239).

von Mayrhauser, A. and Vans, A., (1994). Comprehension Processes During Large Scale Maintenance, ***In Proceedings of the 16th International Conference on Software Engineering***, Sorrento, Italy, (pp. 39-48).

von Mayrhauser, A. and Vans, A., (1995a). Industrial Experience with an Integrated Code Comprehension Model, In ***IEE Software Engineering Journal***, Sept. 1995, (pp. 171 - 182).

von Mayrhauser, A. and Vans, A., (1995b). Program Understanding: Models and Experiments, In M.C. Yovits and M.V. Zelkowitz (Eds), ***Advances in Computers***, Vol. 40, (pp. 1 - 38). Academic Press, Inc

von Mayrhauser, A. and Vans, A., (1996a). On the Role of Program Understanding in Re-engineering Tasks, In ***Proceedings of the 1996 IEEE Aerospace Applications Conference***, (pp. 253 - 267).

von Mayrhauser, A. and Vans, (1996b) A., On the Role of Hypotheses during Opportunistic Understanding While Porting Large Scale Code, In ***Proceedings of the 4th Workshop on Program Comprehension,*** Berlin, (pp. 68 - 77).

von Mayrhauser, A. and Vans, A., (1996c). Identification of Dynamic Comprehension Processes during Large Scale Maintenance, In ***IEEE Transactions on Software Engineering***, vol. 22, no. 6, June 1996, (pp. 424 - 438).

von Mayrhauser, A. and Vans, A., (1997). .Program Understanding Needs During Corrective Maintenance of Large-Scale Software to appear in ***COMPSAC97***.

•