

Comprehension Processes During Large Scale Maintenance

Research Paper

A. von Mayrhauser

A. M. Vans

Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

Abstract

We present results of observing professional maintenance engineers working with industrial code at actual maintenance tasks. Protocol analysis is used to explore how code understanding might differ for small versus large scale code. The experiment confirms that cognition processes work at all levels of abstraction simultaneously as programmers build a mental model of the code. Cognition processes emerged at three levels of aggregation representing lower and higher level strategies of understanding. They show differences in what triggers them and how they achieve their goals. Results are useful for defining core competencies which maintenance engineers need for their work and for documentation and development standards.

1 Introduction

During maintenance and software evolution, software engineers must understand code they haven't written for a variety of tasks.

Existing cognition models [1, 4, 5, 6, 7, 10] emphasize cognition by *what* the program does (a functional approach) and *how* the program works (a control-flow approach). Unfortunately, validation experiments used rather small programs when compared to the cognition needs of most industrial software. Recently, [12, 13] developed an integrated cognition model based on observing industrial programmers at work. We use this model as the basis for deriving processes and information needs.

Table 1 shows characteristics of major experiments with comprehension models. Important characteristics of these experiments are code size, number of subjects, level of expertise, experimental method, and the type of maintenance task. Most of the listed experiments try to validate some component of a model, either directly or indirectly. Some are exploratory and were used to define a model. E.g., the objective in [11] was to determine the characteristics of debug processes that lead to debugging expertise. Table 1 illustrates the difference between our experiments and others in terms of code size, maintenance task, and expertise. Unlike the other observations, we observed experts working with production code. We used protocol analysis to explore whether these maintenance engineers apply the cognition processes described in existing models or whether they needed the integrated

cognition model of [12]. The integrated cognition model emphasizes that understanding is built at all levels of abstraction simultaneously rather than level by level. This necessitates frequent switches between code, design, and application domain knowledge during the cognition process. We report results from an exploratory experiment that identifies dynamics of the cognition process when maintenance engineers work with large operational software products.

Section 2 describes the integrated cognition model with its model components. Section 3 explains our experimental method. This includes participants in the study and experimental procedure (protocol analysis). Protocol analysis is a technique for exploratory research such as ours. We use it to classify model components and to identify information needs and cognition processes.

Section 4 contains the results of the study, focusing on emergent cognition processes. Since cognition is driven by the need for and building of information, we also show the information needs for one of the cognition processes. The integrated model is built around using information at multiple levels of abstraction to construct an understanding of software. Our results also report to which degree cognition processes are simple or multi-level activities and what triggers switches between levels. This becomes important as it prescribes both successful maintenance guidelines as well as tools that support cognitive activities. Interestingly enough, we could aggregate understanding processes at three levels and thus see several classes of strategies emerge. We also interpret the results of our protocol analysis, particularly with respect to maintenance guidelines, preservation of software product information, and core competencies.

2 An Integrated Meta-Model of Code Cognition

Existing program understanding models agree that comprehension proceeds either *top-down*, *bottom-up*, or some combination of the two. Our observations [12] indicate that program understanding involves both top-down and bottom up activities and led to the formulation of a model that integrates existing models as components. This integrated code comprehension meta-model consists of (1.) *Program model*, (2.) *Top-down model*, (3.) *Situation model*, (4.) *Knowledge base*.

<i>Purpose</i>	<i>Code Size</i>	<i>Subjects/Expertise</i>	<i>Language</i>	<i>Experimental Method</i>	<i>Reference</i>
Understand	15 LOC (Module)	94 Novice 45 Grad Students	Pascal	Cloze Test (Fill-in-blank)	Soloway & Ehrlich [9]
Understand	12-42 LOC (Module)	10 Novice 7 Grad Students	Pascal	Cluster Analysis	Rist [6]
Understand	15 LOC (Module)	80 Experts Professional	50% Cobol 50% Fortran	Free-recall & Response time to Comprehension Questions	Pennington [5]
Understand	20 LOC (Module)	52 Novice 18 Intermed. 9 Grad Students	Fortran	Strict recall	Shneiderman [7]
Corrective	400 LOC (Several Modules)	8 Experts 8 Novices	Cobol	Protocol Analysis	Vessey [11]
Enhancement	250 LOC (Several Modules)	4 Experts 2 Intermed.	Fortran	Protocol Analysis	Letovsky [4]
Modification	200 LOC (Several Modules)	40 Experts Professional	50% Cobol 50% Fortran	Comprehension Questions & Protocol Analysis	Pennington [5]
Understand, Corrective, Adaptation, & Enhancement	50-85,000+ LOC (Module-Sys)	11 Experts Professionals	Procedural (e.g. C, C-shell, etc)	Protocol Analysis	von Mayrhauser & Vans [12, 13]

Table 1: **Comprehension Experiments**

The basis for the top-down (also known as *Domain model*) component is Soloway and Ehrlich's [9] top-down model while Pennington's [5] program and situation model components of the meta-model. These three model components reflect mental representations and the strategies used to construct them. They represent views of code at various levels of abstraction. The knowledge component is necessary for successfully building the other three. Thus the complete meta-model describes program, situation, and top-down model building together with the appropriate knowledge for constructing a mental model of the code.

Each model component (as well as the meta-model) builds up knowledge using what is already known about the domain, the architecture or environment, and the particular program. Each model component represents both the internal representation of the program being understood (or *short-term memory*) and a strategy to build this internal representation. The knowledge base furnishes related but previously acquired information. During understanding, new information is chunked and stored into the knowledge base (or *long-term memory*) for future use.

The *Top Down* model is typically active if the code or type of code is familiar. The top-down representation consists of knowledge about the application domain. E.g., a domain model of an Operating System would contain knowledge about OS components (memory management, process management, OS structure, etc.) and interactions among them. This knowledge often includes things like design rationalization (e.g. the pros and cons of First-Come-First-Serve vs. Round Robin scheduling). A new OS will

be easier to understand for someone with this 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 and guide understanding.

When code is completely new to the programmer, Pennington found that the first mental representation programmers build is a *program model* consisting of a control flow abstraction of the program [5]. Suppose an engineer with no OS experience is given an OS to maintain. He may start out by following the flow of control when the only knowledge brought to the task is knowledge of the programming language and standard programming constructs. E.g., a module that represents a scheduling algorithm may contain a doubly linked list to implement a queue for process scheduling, but the engineer may only be able to recognize the linked list. If he has developed a mental representation of *what* the program is doing, statement by statement or in a control flow manner, then he has a program model of the code. Note that a functional understanding of the program is not yet accomplished.

Once the program model representation is constructed, a *situation model* is developed. This representation, also built from the bottom up, uses the program model to create a data-flow/functional abstraction. Using the above example of the engineer with no OS experience, a doubly-linked list recognized as an implementation of a queue for "*process scheduling*" is an element of the situation model. The integrated model also assumes that 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 is

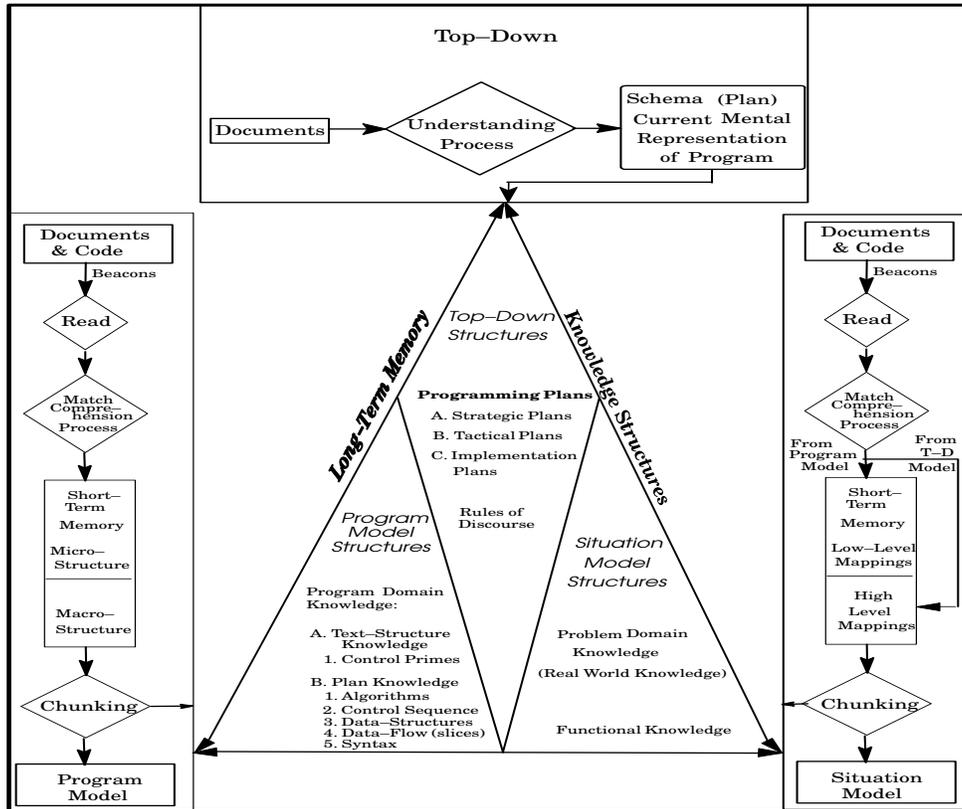


Figure 1: Integrated Code Comprehension Meta-Model

unrealistic. The software products we saw professionals work on (85,000+ lines of code) would create a tremendous cognitive overload.

The *knowledge base*, also known as long-term memory, is usually organized into *schemas* (or *plans*). Schemas are knowledge structures with two parts: slot-types (or templates) and slot fillers. Slot-types describe generic objects while slot fillers are customizations that fit a particular feature. Data structures like lists or trees are examples of slot-types and specific program fragments are examples of slot-fillers. These structures are linked by either a *Kind-of* or an *Is-A* relationship. Schemas are grouped into partitions specifically related to the comprehension processes. For example, knowledge of algorithms is used by the *program model building* process.

A key feature of the integrated meta-model is that any of the three model components may become active at any time during the comprehension process. For example, during program model construction a programmer may recognize a beacon (clue¹) indicating a common task such as sorting. This leads to the hypothesis that the code sorts something, causing a jump to the top down model. The program-

mer then has some objective in mind (e.g. I need to find out whether the sort is in ascending or descending order) and searches the code for clues to support expectations and hypothesis. If, during the search, he finds a section of unrecognized code, he may jump back to program model building. Structures built by any of the three model components are accessible by any other; however, Figure 1 shows that each model component has its own preferred types of knowledge. [12, 13] contain thorough discussions of the integrated meta-model and its component models.

3 Experiments

3.1 Experimental Design

The purpose of our study was to find a code comprehension *process model* using the Integrated Comprehension Meta-model as a guide for large-scale program understanding. We also wanted a high-level preliminary validation through observation. This may ultimately permit validation through controlled experiments. Each observation involved a *programming session* whereby the participants were asked to think aloud while working on understanding code. We audio and 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 found participants with varying degrees of prior experience with the code. This gives the widest degree

¹E.g., a beacon may be the characteristic pattern of value switches that indicate a sort. Or it may be the name of the function (such as QSORT).

of coverage over the code comprehension process.

Eleven subjects have been video- and/or audio-taped, five of which have been transcribed and analyzed. Table 2 defines three major variables for our study. The columns represent *expertise*, the rows represent the amount of accumulated knowledge subjects had acquired *prior* to the start of each observation. The *type of maintenance task* is listed as an entry in the matrix. Each square represents specific observations that are characterized by the row, column, and maintenance task. Abbreviations, for example *C2*, are used in the rest of this paper to identify particular observations.

As the matrix shows (Table 2) these eleven subjects represent good coverage in terms of a varying degree of knowledge about the task and expertise.

Similar to [2], we faced three issues in determining the validity of generalizing results from our data to other maintenance situations. These issues deal with maintenance task, sampling of participants, and external validity of the experimental situation.

1. Task. The code the participants tried to understand and the specific assignment performed were representative of the maintenance tasks encountered in industry. While they were not all doing an identical task, they were all trying to understand industrial code to maintain it. As we begin to understand whether and how cognition differs, we can move to more specialized tasks to explore each situation further.

2. Sampling of participants. How representative are our participants of the larger population of software engineers who work with existing code? There is no reliable answer given the current maturity of the field. We attempted a broad sampling of maintenance tasks, prior work with the code, and programmer experience in domain and language. We make no claim that these protocols represent the full range of cognition behavior of the population of software maintenance engineers. It is more likely that the description of this population will need to be assembled from many (future) studies similar to ours.

3. External validity. This concerns the degree to which the conditions under which our data were collected are representative of those under which actual maintenance occurs. Code cognition for maintenance purposes takes more than two hours. This we considered by including different amounts of prior preparation in our study. All tasks represented actual work assignments. Both strengthen the generalizability of our findings.

3.2 Protocol Analysis

Protocol analysis proceeded in three steps (see Table 3). The following subsections describe the criteria used to classify statements, identify information needs, and analyze protocols for discovery of processes.

3.2.1 Enumeration

The first analysis on the protocols involved *enumeration of action types* as they relate to the integrated

cognition model of [12]. 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 expected actions [11] and searched for them in the transcripts of the protocols. We also analyzed for possible new action types.

<i>Expertise</i> ⇒ Accumulated Knowledge⇓	<i>Language Expert</i>	<i>Domain Expert</i>	<i>Language & Domain Expert</i>
Never Seen Before		C2: Understand Bug	
File Structure Call Graph		C3: Fix Reported Bug	G1: Program: General Understand EN2: Add Function
Requirement & Design Documents	C1: Fix Reported Bug		G2: Understand one Module
Worked some with code, style familiar	L1: Leverage Small Program	C4: Track Down Bug	
Prior code enhancement, debugging, adaptations		AD2: Add Function, Prototype Assess	AD1: Port Program across Platforms
Worked with code several years			EN1: Add Functionality

Table 2: Programming Sessions – All Maintenance Tasks

<i>Enumeration</i>	<i>Segmentation</i>	<i>Process Discovery</i>
Utterance ↓ Action	1. Abstraction level 2. Action types 3. Information Needs	1. Episode level processes 2. Aggregate level processes 3. Session level processes

Table 3: Protocol Analysis Steps

3.2.2 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 domain (top-down), situation, or program model and 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. Included are identification of beacons, domain schemas, hypotheses, strategies, and switching between domain, situation, and program model. For more detail see [12, 13].

Protocol analysis is an iterative, problem solving process. A first pass analysis results in a high-level

<i>Analysis Type</i>	<i>Tag</i>	<i>Action Type</i>	<i>Example Protocol</i>
Action-Type Classification	Sys8	Generate Hypothesis (Program Model)	"...and my assumption is that nil with a little <i>n</i> and nil with a big <i>N</i> are equivalent at the moment."
	Sys7	Chunk & Store knowledge (Program Model)	"So clearly what this does is just flip a logical flag"
<i>Analysis Type</i>	<i>Tag</i>	<i>Information Need Classified As</i>	<i>Example Protocol</i>
Identifying Information Needs		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..."
		Data structure tied to concepts in the domain	"And this looks like some X.25 structure"

Table 4: Example Protocol Analysis – Action Types & Information Needs

classification of programmer actions as either program, situation, or top-down model components of the Integrated Meta-model. This is necessary because similar actions appear in different component processes. For example, hypotheses may be generated while constructing any of the three 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 from the transcribed tapes (see Table 4) or through inference. 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 4, the protocol segment associated with the *Code Block Boundaries* information need indirectly demonstrates that the related domain information would help this programmer understand the code better. Indeed, he spent a great deal of time examining several documents for this very information.

Table 4 contains example protocols to show action type classification and information needs identification. The first half of Table 4 applies only to action type classification. The second half deals with identifying information needs. Column two provides the tag used in action type classification.

3.2.3 Process Discovery

We discovered *dynamic* code understanding *processes* by classifying and analyzing *episodes*. Episodes are single instances of a sequence of action types. An episode starts with a *goal* and embodies the actions to accomplish that goal. For example, determining the function of a specific procedure or routine may entail a sequence of steps that include reading comments, following control flow, and generating questions when a concept is not understood. *Processes* are defined at three different levels; *episodic*, *aggregate*, and *session*

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

levels. Episodes containing common action types with similar goals are defined as a single episodic process. Likewise, common sequences of episodic processes are defined as a single aggregate level process. Finally, the session level process is established by a sequence of similar aggregate level processes.

A process is a sequence of action types, episodes, or aggregates whose purpose is to satisfy a specific goal. We can think of them as three levels of strategies to achieve goals. Specifically, each episode is determined by discovering the goal (or sub-goal) and cataloging all subsequent action types until reaching closure on the goal due to goal satisfaction or goal abandonment. Once episodes are identified, we analyze each, abstract out commonalities, and designate the resulting sequence an episodic process. An aggregate level process emerges from similar episodic processes. Similarly, sequences of aggregate processes are analyzed for commonalities and abstracted into higher-level processes. Once again, common sequences of aggregate level processes produce a single session level process representing a two hour programming session. Section four illustrates these processes using state diagrams.

3.2.4 Study Objectives

1. *The role of Model Components in the Integrated Meta-Model.* We have shown that subjects frequently switch between all model components (i.e. understanding is built at all levels of abstraction simultaneously) [12]. Is there a difference in working at levels of abstraction based on the size of the code under consideration? Answers to this question affect type of knowledge, cognition process, and expertise best suited to large scale code understanding.

2. *Episodes.* We have seen the types of actions engineers execute while working on maintenance tasks [12, 13]. Are there repeated action sequences (episodes) representing lowest level strategies? How often do they occur? How similar are they? Which types of episodes occur most frequently? What information does the engineer need to complete an episode? Do episodes represent understanding at only one level of

		<i>Expertise</i> \Rightarrow <i>Component Size</i> <i>to Understand</i> \Downarrow	<i>Domain</i> <i>Expert</i>		<i>Language</i> <i>& Domain</i> <i>Expert</i>	
<i>Model</i>	<i>Ttl. Num</i>					
Program	334	One Module	C1-Prg	133	G2-Prg	201
Situation	177		C1-Sit	143	G2-Sit	34
Top-Down	78		C1-TD	22	G2-TD	56
Program	204	Several Modules			EN1-Prg	204
Situation	58				EN1-Sit	58
Top-Down	32				EN1-TD	32
Program	71	Whole Program			G1-Prg	71
Situation	41				G1-Sit	41
Top-Down	56				G1-TD	56
Program	74	System			AD1-Prg	74
Situation	39				AD1-Sit	39
Top-Down	212				AD1-TD	212

Table 5: Frequencies of References to Model Components (5 Subjects)

abstraction or do they span all levels?

3. *Aggregate Processes.* If we successfully find episodes, how will they be used in higher level understanding strategies? Are there repeated episode sequences (aggregate processes)? How similar are they? What *triggers* the end of one episode and the beginning of another in a sequence? Are some of these triggers more common than others?

4. *Session-Level Processes.* Once we find the aggregate processes, how will they help in defining the overall maintenance task process (session-level)? Are there repeated sequences of aggregate processes? What distinguishes a switch from one aggregate process and another in the sequence? If we find one process for each type of maintenance task, what are their similarities?

5. *Information Needs and Core Competencies.* Once we find all these processes, can we organize the information engineers need during understanding to support cognition? Can we define core competencies that reflect the minimum knowledge necessary to maintain large scale code?

4 Results

4.1 The Role of Model Components in the Integrated Meta-Model

Table 5 shows an excerpt of the model component analysis for five subjects. The left portion of Table 5 lists the total number of references to the three model components of the integrated meta-model by relative size of the code. We distinguish between small scale: one module ≤ 200 LOC, 200 LOC $<$ several modules $\leq 2,000$ LOC; program size: 2,000 LOC $<$ program $\leq 40,000$ LOC; and large scale, system: $> 40,000$ LOC. Clearly, understanding differs depending on the size of the code. This difference shows itself in the level of abstraction at which a programmer tends to work during understanding. E.g., engineers C1-Prg and G2-Prg had significantly more references to the program model than to the top-down model. For both, the size of the component to understand was a single code module. In contrast, the engineer (AD1) who was porting several programs across operating systems

platforms had almost three times as many references to the top-down model than to the program model.

This table represents a first look at the mental model building strategies maintenance engineers use during understanding. If the size of the component is small enough to understand at a low level of detail, then it makes sense to spend most of the time in the program model. On the other hand, if the code size is large (AD1 worked with a system of about 90,000 lines of code) then understanding occurs preferably at a higher level of abstraction. At the highest level is domain knowledge and ideally we would like to proceed with understanding at this level if the code is large. Obviously then, sufficient knowledge at the domain level is very important for large systems, since programmers refer to this level more during understanding.

While the frequencies of Table 5 indicate how often programmers work at the domain, situation, and program level, this does not describe any comprehension processes.

4.2 Episodes

Episodes are sequences of action types carried out to accomplish a goal. Episodes containing common actions with similar goals emerge as the lowest level processes. Each discovered episode is specified by its action sequences. To illustrate, in our example protocol, episodic process *P1 - Read Block in Sequence* starts with the overall goal of understanding a specific block of code, e.g. "I'm going to read the description and see if it gives me some good clues as to what's going on." Some of the observed actions that support the original goal are: generating hypotheses while reading comments, chunking information, making note of interesting aspects, and postponing investigation of them.

Table 6 lists seven episodic processes and how often they occurred in one example transcript (subject G2 (see Table 2)). It shows that the subject spent the majority of his time reading the code, determining the behavior of a variable, and incorporating this knowledge into his mental model of the program module. The engineer applied a *systematic strategy* of reading

each line of code in approximate sequence. Figure 2 presents this process in graph form as a state machine to illustrate the basic form of episodes. Arcs indicate action types while states represent level of understanding.

Table 6 also shows that preferred episodic processes exist. Processes P1 and P3 were the most frequent. These two processes are preferred during detailed understanding of one module. Based on Table 5, we expect to find other processes, such as P2, more frequently referenced when understanding larger code segments.

<i>Episodic Process Name</i>	<i>Code</i>	<i>Number</i>
Read Block in Sequence	P1	7
Integrate Not Understood	P2	4
Determine Variable Def/Use	P3	7
Incorporate Acquired Program Knowledge	P4	5
Identify Block Boundaries	P5	2
Resolve Deferred Questions	P6	2
Understand a Procedure Call	P7	1

Table 6: **Episodic Process Frequency Count (1 Subject)**

Individual episodes can vary greatly, because their goals are very different. An episode may use the same action types as another but occur in a different order. As we combine and abstract commonalities at different process levels, the processes themselves become similar. This corresponds to similar higher level goals which the aggregate and session level processes support. However, they use different episodes as their low level tactics. Two different action sequences characterized as a P1 episodic process illustrate different traces through the P1 state diagram of Figure 2.

1. Examine next module in sequence → Chunk & Store knowledge → Examine next module in sequence → Generate hypothesis.
2. Read Code Comments → Generate Hypothesis → Chunk & Store → Note Interesting Identifiers & Determine Key aspects → Chunk & Store → Read next module in sequence → Note Interesting Identifiers & Determine Key aspects → Chunk & Store ...

During our analysis we were able to associate information needs (and their frequencies) with action types and thus with episodic processes. Table 7 shows information needs for process P1 for subject G2. The three most frequently needed information types for P1 directly relate to the activities shown in the state diagram. E.g. determining the end-of-block condition requires code block boundary information.

Interestingly, we could not find processes that occurred on a single level of abstraction and therefore classified as purely top-down, situation, or program model processes. Many episodes contained actions that were associated with all three integrated model components. This supports the idea that programmers constantly switch between model components

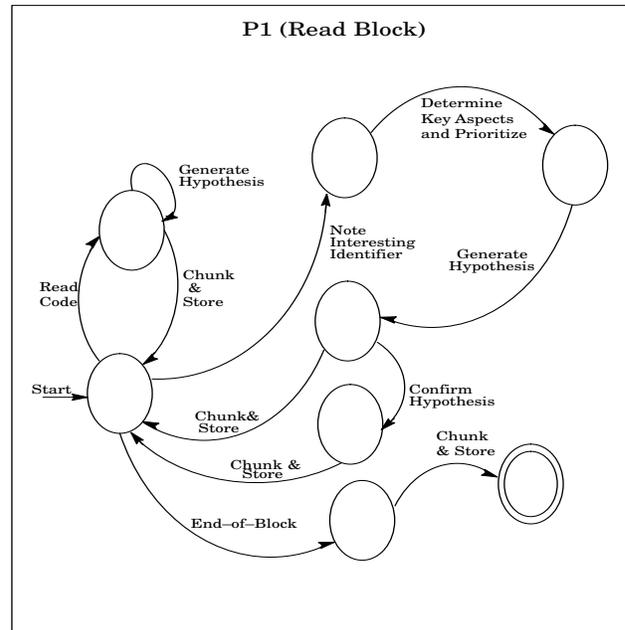


Figure 2: **Episodic Process – Read Block**

(levels of abstraction). In this, episodic processes are similar to the design process reported in [2], which isn't too surprising since cognition can be considered a recreation of the design task. These results also show that for cognition aids to be effective, tools or documentation must support work at all levels of abstraction and the frequent switches between them.

<i>Episodic Process</i>	<i>Information Needs</i>	<i>Number</i>
P1: Read Block	Code Block Boundaries	4
	Data Type definitions & location of identifiers	3
	Call Graph Display	2
	History of past modifications	1
	Data structure definitions tied to domain concepts	1
	Location of called procedures	1
	History of browsed locations	1
	Beacons tied to situation model or program model	1
	Description of system calls	1
	Location of documents for program & domain	1
		1
		1
		1

Table 7: **Understanding One Module – Information Needs for Process P1**

4.3 Aggregate Processes

Three aggregate processes were discovered in G2's protocol and we illustrate one below in the form of a state diagram. Table 8 shows that aggregate processes consist of episodic processes.

Table 9 shows frequencies of aggregate level processes. At the aggregate level, processes PA, PB, and PC begin to look very similar. One conjecture is that

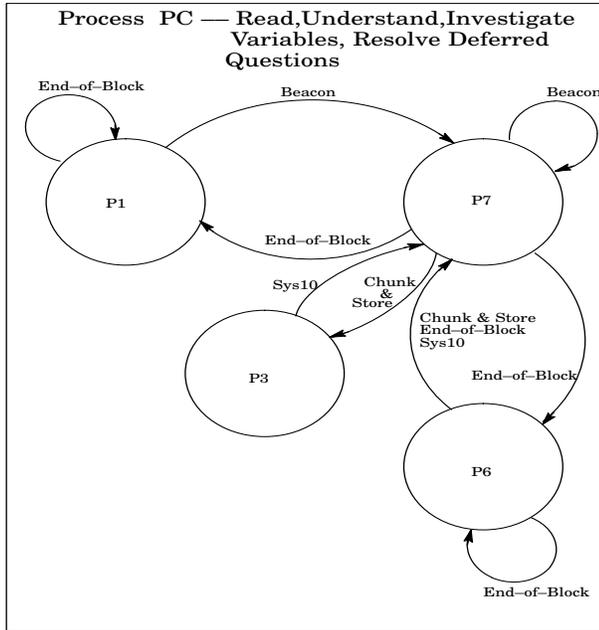


Figure 3: Process PC – Aggregate-Level

these aggregate processes represent instances of a similar higher level strategy.

Episodic Process	PA	PB	PC
P1: Read Block in Sequence	X	X	X
P2: Integrate Not Understood	X		
P3: Determine Variable Def/Use	X	X	X
P4: Incorporate Acquired Program Knowledge		X	
P5: Identify Block Boundaries		X	
P6: Resolve Deferred Questions			X
P7: Understand a Procedure Call			X

Table 8: Aggregate Processes – Episodic Composition

Subject G2 demonstrated a systematic approach [3] to understanding. Another common strategy used for code comprehension is opportunistic, one in which the programmer follows individually determined, *relevant* threads through code and documentation. For this type of strategy we might not see the nicely organized processes we saw with systematic understanding. Instead, we may have to deal with several parallel processes. E.g., an action deferment in the systematic process becomes a process switch in the opportunistic process. In this case, we might see interleaved pieces of parallel processes.

Triggers cause state changes between processes. They can be code induced (e.g. end of code block) or an action type (e.g chunk and store knowledge). Table 10 lists the triggers found in the example protocol and their frequencies for Process PC and the total for all the aggregate processes. Beacons and end-of-block triggers were the most frequent triggers. Again, this could be a by-product of the systematic strategy used

by this subject. E.g., a jump out of episodic process P1 (Read Block in Sequence) into process P7 (Understand Procedure Call) is caused when G2 encounters an unrecognized procedure call (a beacon). He decides to understand what the procedure does. After investigating it he reaches its end (End-of-block) which triggers the end of P7 and resumption of P1 where he last left off.

Aggregate Process Name	Code	Number
Read, Integrate, Investigate variables	PA	4
Read, Incorporate acquired program knowledge, Investigate variables, Identify Block Boundaries	PB	3
Read, Understand, Investigate variables, Resolve deferred questions	PC	3

Table 9: Aggregate Process Frequency Count (1 Subject)

Process Trigger	Frequencies	
	PC	All Aggregate Procs
Beacon	7	14
Chunk & Store Knowledge	2	8
End-of-Block	7	20
End-of-Stack	0	2
Understanding strategy determined (Sys10)	2	7

Table 10: Process Trigger Frequencies (1 Subject)

4.4 Session Level Processes

Session level processes are at the highest level. The state diagram in Figure 4 was derived in the same way as the aggregate-level processes by tracking the sequences of aggregate-level processes. This diagram represents a general understanding maintenance task. At the highest level, only “End-of-block” and “Chunk & Store” cause switches from one aggregate-level process to the next.

The session-level process (for Understanding a single module) shows that all the aggregate-level processes represent investigation towards building chunks [5, 8]. Chunking is an important abstraction mechanism in code understanding. Thus, at the session-level the purpose of each aggregate process is to understand a block of code (using different detail steps and information) and then to chunk and store the learned information.

4.5 Information Needs

Our analysis found a variety of information needs and useful tool capabilities [12]. We can abstract and describe the evidence as the minimum knowledge necessary to perform maintenance tasks and partition the knowledge into groups of program, situation, and top-down related structures. Such knowledge could be defined as *core competencies*. While our results are incomplete, we have an indication of the types of information that will be useful for maintenance engineers. Table 11 summarizes this information.

<i>Domain Information Necessary</i>	
Product Specific Knowledge	Commands and Use (e.g. HP/UX vs. UNIX operating system commands) System Configuration (e.g. How to configure the system for test or bug reproduction.)
Area Knowledge	Standard, product independent information (e.g. Operating systems principles.) Prior Experience Formal instruction
Architecture	Structure (e.g. An operating system as components: Process, I/O, memory, and file management.) Interconnections (e.g. How are they related.)
Cross references	Within the domain including where to find the information we need (e.g. the expert or specific text book.)
Key terms	VERY IMPORTANT This guides understanding and ties the domain model to the situation and program models. (e.g. PROCESS MANAGER using ROUND ROBIN scheduling algorithm.)
<i>Situation Information Necessary</i>	
Algorithms and data structures	Language independent, detail design level (e.g. functional sequence of steps in the round robin algorithm or a graphical representation of a process queue.)
Detailed design	Close to code, but language independent. Specific product information in functional terms.
“Pop-up” technology connected to domain information	Find design rational. Connect algorithm to purpose of application.
Conventions	Use the same terms across comprehension & models. THESE ARE THE KEY TERMS!
Cross reference levels of information with connections to other models	Also within the same situation model level
<i>Program Model Information Necessary</i>	
Variable & component names	KEY TERMS : meaningful mnemonic and acronyms for symbols.
“Pop-ups” connected to situation and domain	Capability to follow beacon to design and domain information.
Critical sections of code identified	Focus attention; improve efficiency
Formalized beacons	Focus attention; improve efficiency
Cross references & Connections	Back to the situation and domain levels.

Table 11: Core Competencies by Model Component

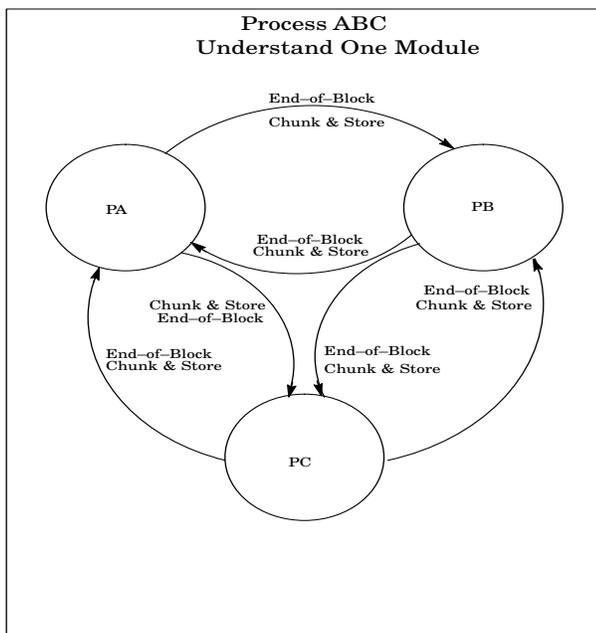


Figure 4: Process ABC – Session Level

While much of this information may already exist (buried in magnificent tomes), programmers could benefit from capabilities for focused access to answer specific questions. The problems with much of today’s information are long access times and information confusion. The ability to focus and construct documentation around cognition processes are likely to be more successful than ever more volumes of documentation. This also means we must reorganize and present information so that it supports the cognitive processes we found. If we analyze the episodic, aggregate, and session level cognitive processes for tool requirements, it is quite obvious that few tools exist that support even a part of these processes. At the heart of the difficulty is the need of programmers to work with a variety of levels and types of information when most of our tools and documentation are single level entities (e.g. a design document, a user manual, a book on Operating Systems concepts, etc.).

Support in connecting information at all levels of understanding appears promising. Simple beginnings are development standards that carry key terms throughout a project. (E.g. when trying to find out what “X.25” means in the code, one can search for the same term in the detailed design, design architecture, and domain level documentation. In this case, “X.25”

is the focus, but one may need to know how it relates to domain level knowledge. For this question, a cross-reference list of where and how often a variable called "X.25" is used in the code is useless.)

5 Conclusion

Program understanding is a key factor in software maintenance and evolution. This paper reported on an experiment with industrial programmers to discover the comprehension processes and the supporting information that programmers use when trying to understand production code. While our sample was small, this exploratory experiment showed several interesting results:

1. Programmers use a multi-level approach to understanding, frequently switching between program, situation, and domain (top-down) models. Effective understanding of large-scale code needs significant domain information.
2. Maintenance activities can be described by a distinct small set of cognition processes. These aggregate into higher level processes, each with their goals and information needs.
3. Current practice of documentation and coding does not encourage efficient understanding as it compartmentalizes knowledge by type of document and rarely provides the cross references that are needed to support programmers' cognitive needs.

We hope that our results stimulate further experiments to increase our knowledge of industry-size code cognition. Then we can base maintenance processes on the cognition needs of people.

Acknowledgement

Part of this work was funded through a grant from the Hewlett-Packard Co., Inc.

References

- [1] Ruven Brooks, "Towards a theory of the cognitive processes in computer programming", In: *International Journal of Man-Machine Studies*, Vol. 9, pp. 737-751, 1977.
- [2] Raymonde Guindon, Herb Krasner, and Bill Curtis, "Breakdowns and Processes During the Early Activities of Software Design by Professionals", In: *Empirical Studies of Programmers: Second Workshop*, Eds. Olson, Sheppard, and Soloway, Ablex Publishing Corporation, pp. 65 - 82, ©1987.
- [3] Jurgen Koenemann and Scott P. Robertson, "Expert Problem Solving Strategies for Program Comprehension", In: *CHI'88*, pp. 125-130, March 1991.
- [4] Stanley Letovsky, "Cognitive Processes in Program Comprehension", In: *Empirical Studies of Programmers, First Workshop*, Eds. Soloway and Iyengar, Ablex Publishing Corporation, pp. 58 - 79, ©1986.
- [5] Nancy Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs", In: *Cognitive Psychology*, Vol. 19, pp.295-341, 1987.
- [6] Robert S. Rist, "Plans in Programming: Definition, Demonstration, and Development", In: *Empirical Studies of Programmers: 1st Workshop*, Eds. Soloway and Iyengar, Ablex Publishing Corporation, pp. 28-47, ©1986.
- [7] Ben Shneiderman, "Exploratory Experiments in Programmer Behavior", In: *International Journal of Computer and Information Sciences*, Vol. 5, No.2, pp. 123-143, 1976.
- [8] Ben Shneiderman, *Software Psychology, Human Factors in Computer and Information Systems*, In: Chapter 3, Winthrop Publishers, Inc., pp. 39-62, ©1980.
- [9] Elliot Soloway and Kate Ehrlich, "Empirical Studies of Programming Knowledge", In: *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, pp. 595-609, September 1984.
- [10] Elliot Soloway, Beth Adelson, and Kate Ehrlich, "Knowledge and Processes in the Comprehension of Computer Programs", In: *The Nature of Expertise*, Eds. M. Chi, R. Glaser, and M.Farr, Lawrence Erlbaum Associates, Publishers, pp. 129-152, ©1988.
- [11] Iris Vessey, "Expertise in debugging computer programs: A process analysis", In: *International Journal of Man-Machine Studies*, Vol. 23, pp.459-494, 1985.
- [12] A. von Mayrhauser and A. Vans, "From Program Comprehension to Tool Requirements for an Industrial Environment", In: *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, pp. 78 -86, July 1993.
- [13] A. von Mayrhauser and A. Vans, "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, July 1993.