Program comprehension during software maintenance and evolution

Article *in* Computer · September 1995

DOI: 10.1109/2.402076 · Source: IEEE Xplore

Program Comprehension During Software Maintenance and Evolution

Anneliese von Mayrhauser
A. Marie Vans
Colorado State University

Code cognition models examine how programmers understand program code. The authors survey the current knowledge in this area by comparing six program comprehension models.

Program understanding is a major factor in providing effective software maintenance and enabling successful evolution of computer systems. For years, researchers have tried to understand how programmers comprehend programs during software maintenance and evolution. Five types of tasks are commonly associated with software maintenance and evolution: adaptive, perfective, and corrective maintenance; reuse; and code leverage. Each task type typically involves certain activities (see Table 1). Some activities, such as understanding the system or problem, are common to several tasks. To analyze the cognitive processes behind these tasks, researchers have developed several models (see Table 2).

In this article, we describe common elements of six cognition models and compare them based on their scope and experimental support. All models use existing knowledge to build new knowledge about the mental model of the software that is under consideration. Many of these models are based on exploratory experiments, and some of those models have been validated. Programmers employ various strategies and use cues in code or documentation as guidance. However, their level of expertise greatly influences how efficiently they understand the code.

Because of limited knowledge about specialized cognition needs for some maintenance tasks, the code cognition models do not represent every single task listed in Table 1. Most models assume that the objective is to understand all of the code rather than a particular purpose, such as debugging. While these general models can foster a complete understanding of a piece of code, they may not always apply to specialized tasks that more efficiently employ strategies geared toward partial understanding.

We identify open questions, particularly considering maintenance and evolution of large-scale code. These questions relate to scalability of existing experimental results with small programs, validity and credibility of results based on experimental procedure, and challenges of data availability.

COMMON ELEMENTS OF COGNITION MODELS

The program comprehension process uses existing knowledge to acquire new knowledge that ultimately meets the goals of a code cognition task. This process references both existing and newly acquired knowledge to build a mental model of the software that is under consideration. Understanding depends on strategies. While these cognition strategies vary, they all formulate hypotheses and then resolve, revise, or abandon them.

Knowledge

Programmers possess two types of knowledge: general knowledge that is independent of the specific software application they are trying to under-

0018-9162/95/$4.00 © 1995 IEEE

stand and software-specific knowledge that represents their level of understanding of the software application. During the understanding process, they acquire more software-specific knowledge but may also need more general knowledge—for example, how a round-robin algorithm works. Existing knowledge concerns the programming languages and computing environment, programming principles, domain-specific architecture choices, algorithms, and possible solution approaches. If the programmer has worked with the code before, existing knowledge includes any (partial) mental model of the software.

New knowledge primarily concerns the software product. It is acquired throughout the code understanding process. This knowledge relates to functionality, software architecture, the way algorithms and objects are implemented, control, dataflow, and so on. It obviously spans many abstraction levels from "this is an operating system" to "variable *q* is incremented in this loop." The understanding process matches existing knowledge with software knowledge until the programmers believe they understand the code. The set of matches is the mental model; it can be complete or incomplete.

Table 1. Tasks and activities requiring code understanding.

Maintenance tasks	Activities
Adaptive	Understand system Define adaptation requirements Develop preliminary and detailed adaptation design Code changes Debug Regression tests
Perfective	Understand system Diagnosis and requirements definition for improvements Develop preliminary and detailed perfective design Code changes/additions Debug Regression tests
Corrective	Understand system Generate/evaluate hypotheses concerning problem Repair code Regression tests
Reuse	Understand problem, find solution based on close fit with reusable components Locate components Integrate components
Code leverage	Understand problem, find solution based on predefined components Reconfigure solution to increase likelihood of using predefined components Obtain and modify predefined components Integrate modified components

Mental model

The mental model is an internal, working representation of the software under consideration. It contains static entities such as text structures, chunks, plans, hypotheses, beacons, and rules of discourse. Top-level plans refine into more detailed plans or chunks. Each chunk, in turn, represents a higher level abstraction of other chunks or text structures. A chunk's construction combines several dynamic behaviors, including strategies, actions, episodes, and processes.

Table 2. Code cognition models.

Model	Maintenance activity	Authors
Control-flow	Understand	Pennington[1]
Functional	Understand	Pennington[1]
Top-down	Understand	Soloway, Adelson, and Ehrlich[2,3] Rist[4]
Integrated	Understand, corrective, adaptive, and perceptive	Von Mayrhauser and Vans[5]
Other	Enhancement	Letovsky[6]
	Corrective	Vessey[7]
	Understand	Brooks[8] Shneiderman and Mayer[9]

Static elements of the mental model

Text-structure knowledge includes the program text and its structure. Text-structure knowledge for understanding grows through experience and is stored in long-term memory. Pennington[1] uses text-structure knowledge to explain control-flow knowledge for program understanding. Structured programming units form text structure and organize knowledge.

Examples of text-structure knowledge units include control primes—iteration (loop constructs), sequences, and conditional constructs (for example, If-Then-Else); variable definitions; module calling hierarchies; and module parameter definitions. The program text's microstructure contains the actual program statements and their relationships. For example, a Begin statement starts a block of code, while a subsequent If statement indicates a conditional control structure with a particular purpose. Their relationship is such that the If is part of the block initiated by the Begin.

Chunks are knowledge structures containing various levels of text-structure abstractions. Text-structure chunks are called macrostructures, which are identified by a label and correspond to the program text's control-flow organization.[1] For example, the microstructure for a sort includes all its statements, while the macrostructure is an abstraction of the block of code and includes only the label sort. Lower level chunks can form higher level chunks. Higher level chunks comprise several labels and the control-flow relationships between them.

Plans are knowledge elements for developing and validating expectations, interpretations, and inferences; they capture the comprehender's attention during the program understanding task. These plans also include causal knowledge about the information flow and relationships between parts of program. Plans are schemas (frames) with two parts: slot types (templates) and slot fillers. Slot types describe generic objects, while slot fillers are customized to fit a particular feature. Data structures such as 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.

Programming plans can be high-, low-, or intermediate-level programming concepts. For example, searching, sorting, and summing algorithms as well as data-structure knowledge such as linked-lists and trees are intermediate-level concepts. Iteration and conditional code segments are low-level concepts. Programming plan knowledge includes roles for data objects, operations, tests, other plans, and constraints on what can fill the roles.

Domain plans incorporate all knowledge about the problem area except for code and low-level algorithms. Domain plans apply to objects in the real world. For instance, plans to develop a software tool for designing automobiles would include schemas related to the function and appearance of a generic car. An appropriate plan would also include slots for problem domain objects such as steering wheels, engines, doors, and tires. Domain plans are crucial for understanding program functionality. Control-flow plans alone are not enough to understand aspects such as causal relationships among variables and functions. Domain plans also address the environment surrounding the software application, the domain-specific architecture, and solution alternatives.

Letovsky[6] refers to *hypotheses* as conjectures and defines them as results of comprehension activities (actions) that can take seconds or minutes to occur. Letovsky has identified three major types of hypotheses:

- *why* conjectures hypothesize the purpose of a function or design choice;
- *how* conjectures hypothesize the method for accomplishing a program goal; and
- *what* conjectures hypothesize classification—for example, a variable or function.

Degrees of certainty associated with a conjecture vary from uncertain guesses to almost certain conclusions.

Brooks[8] theorizes that hypotheses are the only drivers of cognition. Understanding is complete when the mental model contains a complete hierarchy of hypotheses. At the top is the primary hypothesis: a high-level description of the program function, which is necessarily global and nonspecific. Subsidiary hypotheses support the primary hypothesis. Hypotheses can fail for three reasons: code to verify a hypothesis can't be found; confusion exists because one piece of code satisfies different hypotheses; or code can't be explained.

Hypotheses drive the direction of further investigation. Generating hypotheses about code and investigating whether they hold or must be rejected is an important facet of code understanding.

Dynamic elements of the mental model

A strategy guides the sequence of actions while following a plan to reach a particular goal. For example, if the goal is to understand a block of code, the strategy might be to systematically read and understand each line of code while building a mental representation at higher and higher abstraction levels. An opportunistic strategy studies code in a more haphazard fashion.

Strategies also differ in how they match programming plans to code. Shallow reasoning[2,3] does so without in-depth analysis. Many experts do this when they recognize familiar plans. Deep reasoning[2,3] looks for causal relationships among procedures or objects and performs detailed analyses.

Strategies guide two understanding mechanisms that produce information: chunking and cross-referencing. *Chunking* creates new, higher level abstraction structures from chunks of lower level structures. As structures are recognized, labels replace the detail of the lower level chunks. In this way, lower level structures can be chunked into larger structures at a higher abstraction level. For example, a code segment may represent a linked-list definition as pointers and data. In an operating system definition, this may be abstracted as a ready queue. The code segment that takes a job from the ready queue, puts it into the running state, monitors elapsed time, and removes the job after expiration of the time quantum may be abstracted as a round-robin scheduler. The code for the queue, timer, and scheduling are microstructures. Continued abstraction of the round-robin scheduler, dead-lock resolution, interprocess communication, process creation/deletion, and process synchronization eventually leads to the higher level structure definition, "process management of the operating system."

Cross-referencing relates different abstraction levels, such as a control-flow view and a functional view, by mapping program parts to functional descriptions. For instance, recognizing that a code segment manages processes and determining its purpose says something about functionality. Hence, cross-referencing is essential to building a mental representation across abstraction levels.

Code cognition formulates hypotheses, checks whether they are true or false, and revises them where necessary. Hypotheses, like plans and schemas, exist at all levels of abstraction. The key is to keep the number of open hypotheses manageable while increasing understanding.

Facilitating knowledge acquisition

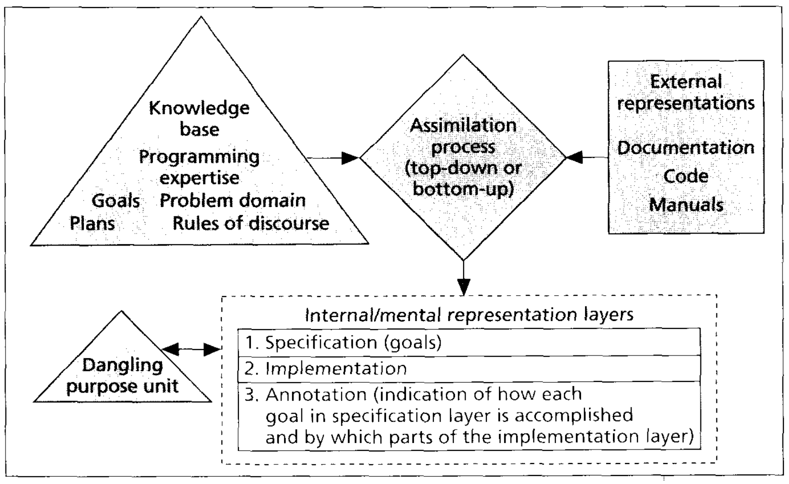
Beacons, cues that index into knowledge, can be text or a component of other knowledge. For example, a swap statement inside a loop or a procedure can be a beacon for a sorting function; so can the procedure name Sort. Beacons are useful for gaining a high-level understanding.[8]

Rules of discourse are conventions in programming such as coding standards, algorithm implementations, expected use of data structures, and so on. Rules of discourse set programmer expectations. Programming plans are retrieved from long-term memory via these expectations. Soloway, Adelson, and Ehrlich[2] show that expert programmers perform significantly better on plan-like code (code fragments that match expert programming plans) than on nonplan-like code. In practice, this means that unconventional algorithms and programming styles are much harder to understand, even for experts.

Expert characteristics

A programmer's level of expertise in a given domain greatly affects program understanding. Experts tend to show the following characteristics:

- They organize knowledge structures by functional characteristics of the domain in which they are experts. For instance, novices might understand a particular program organized according to the program syntax. But experts might organize program knowledge in terms of algorithms rather than the syntax used to implement the program.[10]
- Experts have developed efficiently organized specialized schemas,[10] often abstracted from previously designed software systems.
- Specialized schemas contribute to efficient problem decomposition and comprehension.[10] Top-down comprehension becomes feasible for problems that match specialized schemas.
- Experts approach problem comprehension with flexibility.[7] They discard questionable hypotheses and assumptions much more quickly than novices do, and they tend to generate a breadth-first view of the program. As more information becomes available, they refine their hypotheses.



Knowledge base
Programming expertise
Goals
Problem domain
Plans
Rules of discourse
Assimilation process (top-down or bottom-up)
External representations
Documentation
Code
Manuals
Internal/mental representation layers
1. Specification (goals)
2. Implementation
3. Annotation (indication of how each goal in specification layer is accomplished and by which parts of the implementation layer)
Dangling purpose unit

Figure 1. Letovsky comprehension model.

COGNITION MODELS

Common elements of program cognition models occur in various theories. We discuss the most important ones below.

Letovsky model

Letovsky's high-level comprehension model (see Figure 1) has three main components: a knowledge base, a mental model (internal representation), and an assimilation process.[6] The knowledge base contains programming expertise, problem-domain knowledge, rules of discourse, plans, and goals.

The mental model has three layers: a specification, an implementation, and an annotation layer. The specification layer—the program's highest abstraction level—completely characterizes the program goals. The implementation layer contains the lowest level abstraction, with data structures and functions as entities. The annotation layer links each goal in the specification layer to its realization in the implementation layer. However, these links can be incomplete. The dangling-purpose unit models such unresolved links.

The assimilation process occurs either top-down or bottom-up. It is opportunistic in that the programmers proceed in the way they think yields the highest return in knowledge gain. To contribute to one of the three layers constructed in the mental representation, the understanding process matches code, documents, and so on with elements from the knowledge base.

Shneiderman and Mayer model

Shneiderman and Mayer's comprehension model (see Figure 2)[9] recodes the program in short-term memory into an internal semantic representation via a chunking process. These internal semantics contain different levels of program abstraction. At the top are high-level concepts such as program goals. The lowest levels contain details such as the algorithms used to achieve program goals.

Long-term memory, a knowledge base with semantic and syntactic knowledge, assists during internal semantics construction. Syntactic knowledge is programming language dependent, while semantic knowledge concerns general programming knowledge independent of any specific programming language. Like working memory, semantic knowledge in long-term memory is layered and incorporates high-level concepts and low-level details. Program understanding is directional: It starts with the program code and continues until the problem statement is reconstructed.

Brooks model

Brooks[8] sees program comprehension as the reconstruction of the domain knowledge used by the initial developer. Understanding involves recreating the mappings from the problem domain into the programming domain, through several intermediate domains.

The problem (application) domain concerns real-world problems—for example, maintaining appropriate inventory levels. The objects are inventories, physical entities

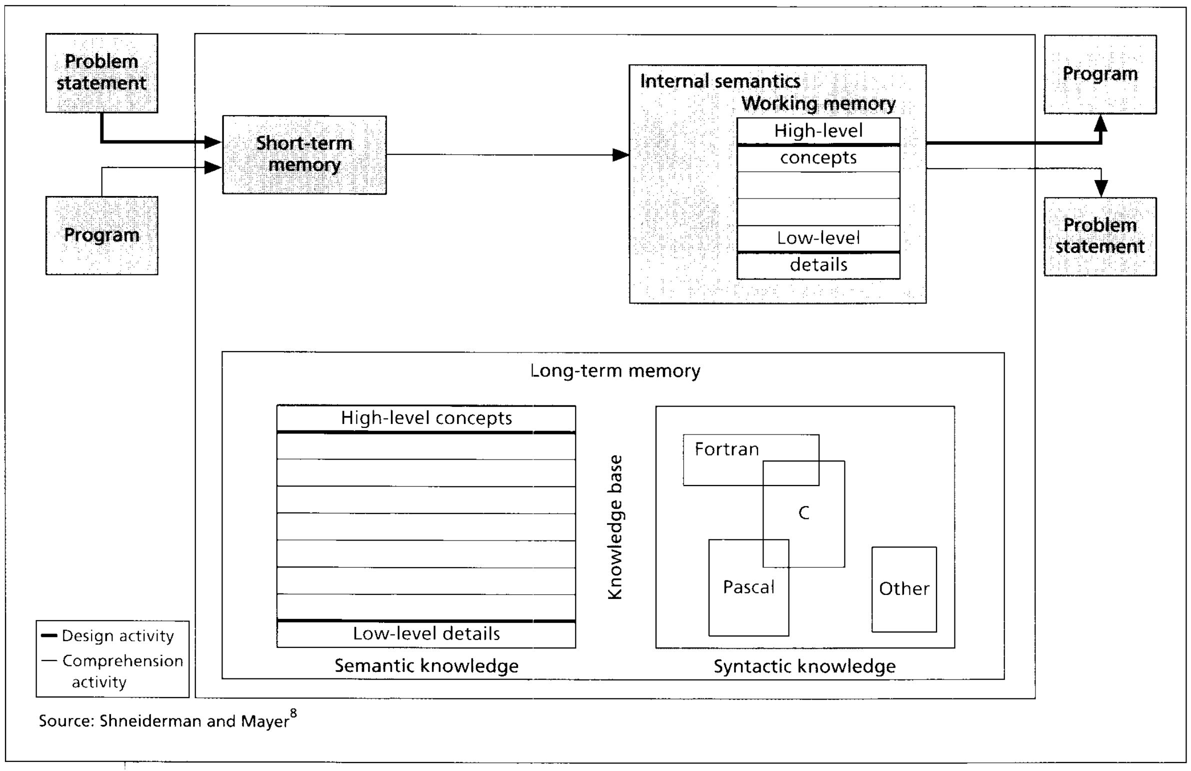


Figure 2. Shneiderman and Mayer comprehension model.

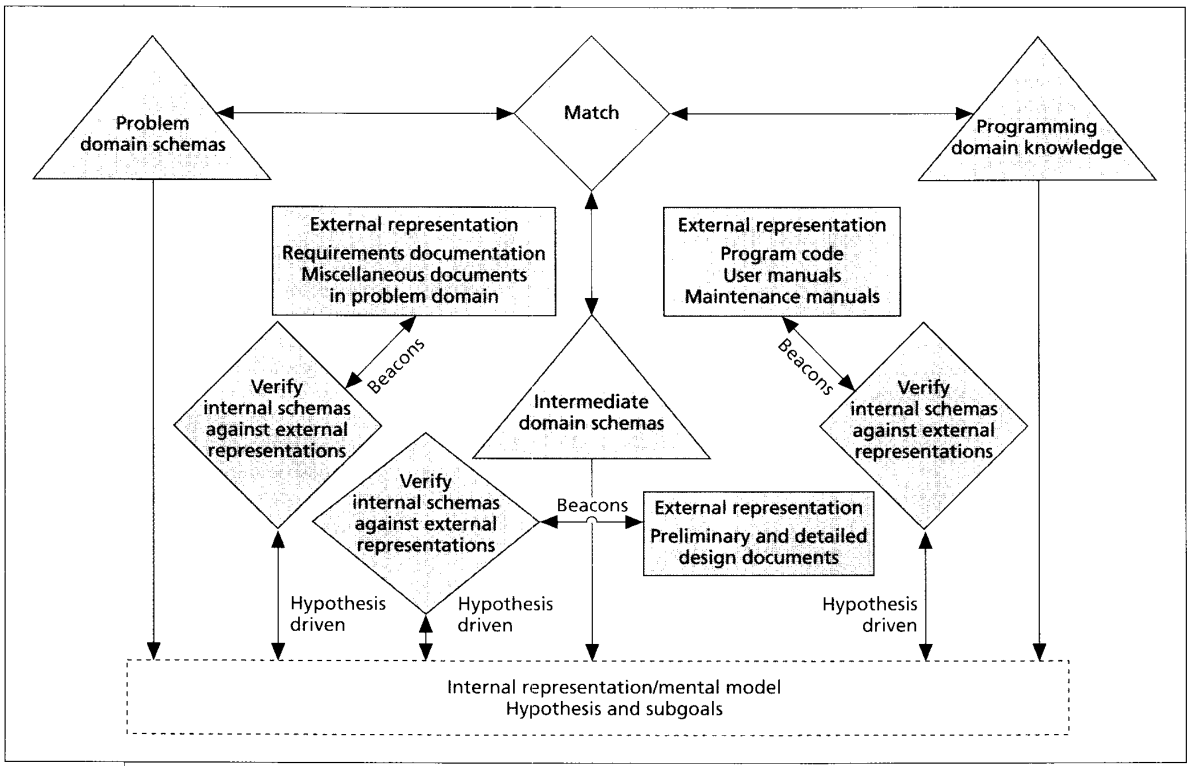
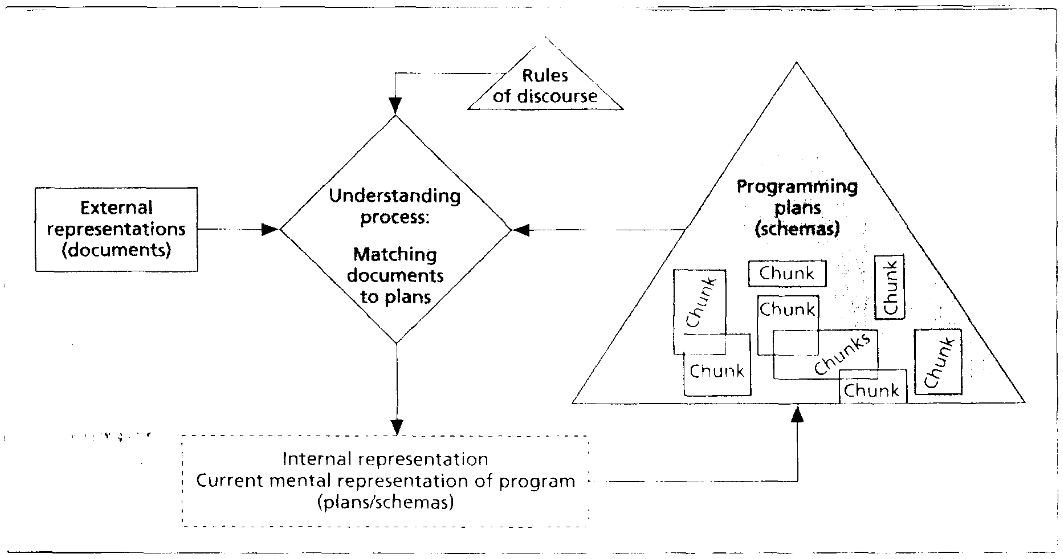

Problem domain schemas
Match
Programming domain knowledge
External representation
Requirements documentation
Miscellaneous documents in problem domain
External representation
Program code
User manuals
Maintenance manuals
Beacons
Beacons
Verify internal schemas against external representations
Intermediate domain schemas
Verify internal schemas against external representations
Verify internal schemas against external representations
Beacons
External representation
Preliminary and detailed design documents
Hypothesis driven
Hypothesis driven
Hypothesis driven
Internal representation/mental model
Hypothesis and subgoals

Figure 3. Brooks comprehension model.



Rules of discourse
External representations (documents)
Understanding process: Matching documents to plans
Programming plans (schemas)
Chunk
Chunk
Chunk
Chunk
Chunks
Chunk
Chunk
Chunk
Internal representation
Current mental representation of program (plans/schemas)

Figure 4. Soloway, Adelson, and Ehrlich comprehension model.

that have properties such as size, cost, and quantity. Once the physical objects are characterized, intermediate knowledge domains are required. Inventory cost might include not only the actual cost but also overhead such as storage. Accounting knowledge recognizes the appropriate overhead calculations. Knowledge of program syntax helps implementation. This example uses at least four different knowledge domains to reach the programming domain: inventories, accounting, mathematics, and programming languages.

Knowledge within each domain includes details about domain objects, the operations allowed on them, and the order in which those operations are allowed. Interdomain knowledge describes the relationships between objects in different but closely related domains, such as the relationship between the general operating system domain and the Unix operating system domain.

The mental model is built through a top-down process that successively refines hypotheses and auxiliary hypotheses. Hypotheses are iteratively refined, passing through several knowledge domains until they can be matched to specific code in the program or a related document.

Figure 3 illustrates the Brooks model. Knowledge, shown as triangles, can be used directly for hypothesis generation in the mental model or matched (mapped) from one domain to another. A cognitive process verifies that internal representations reflect knowledge contained in external representations such as code, design documents, or requirements specifications. Beacons are the main vehicle for this verification, which is also hypothesis driven. Once a hypothesis is generated the external (internal) representations can be searched to support the hypothesis.

Top-down model: Soloway, Adelson, and Ehrlich

Top-down understanding (see Figure 4)[2,3] typically applies when the code or type of code is familiar. The code can then be decomposed into the typical elements of that system type. Theoretically, new code could be understood entirely in a top-down manner if the programmer had already mastered code that performed the same task and was structured in exactly the same way.

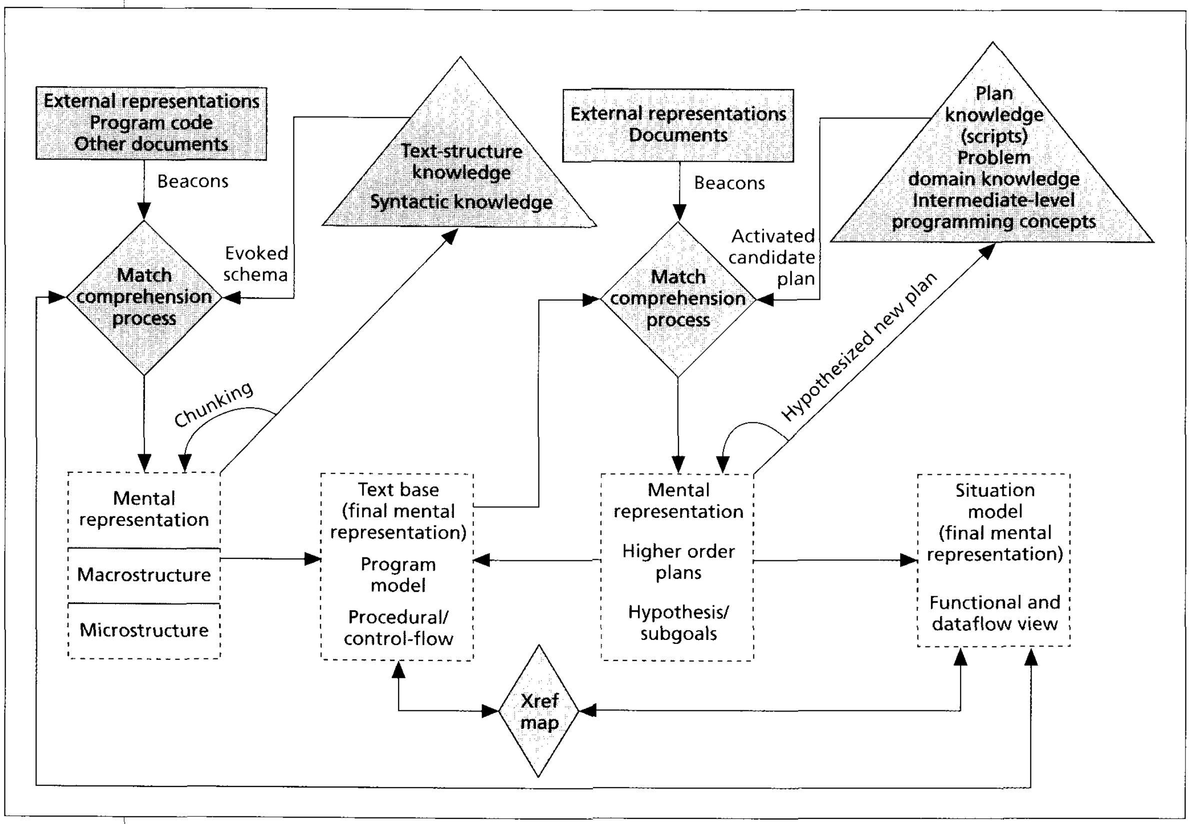
This model uses three types of plans: strategic, tactical, and implementation. Strategic plans describe a global strategy used in a program or algorithm and specify language-independent actions. Tactical plans are local strategies for solving a problem; these plans contain language-independent specifications of algorithms. Implementation plans are language dependent and are used to implement tactical plans; they contain actual code fragments.

A mental model is constructed top-down. It contains a hierarchy of goals and plans. Rules of discourse and beacons help decompose goals into plans and plans into lower level plans. Typically, shallow reasoning builds the connections between the hierarchical components.

Figure 4 shows the model's three major components. The triangles represent knowledge (programming plans or rules of discourse). For example, a subset of the rules of discourse might include the following:

- variables updated same way as initialized,
- no dead code,
- a test for a condition means that the condition must be potentially true,
- don't do double duty with code in a nonobvious way, and
- an If is used when a statement body is guaranteed to execute only once; a While is used when the statement may need to be executed repeatedly.

The diamond represents the understanding process. The rectangles illustrate internal or external program representations. (External representations include documents such as requirements or design documents; code; user, reference, or maintenance manuals; and other mis-



External representations Program code Other documents
Beacons
Match comprehension process
Evoked schema
Text-structure knowledge Syntactic knowledge
Chunking
Mental representation
Macrostructure
Microstructure
Text base (final mental representation)
Program model
Procedural/ control-flow
External representations Documents
Beacons
Activated candidate plan
Match comprehension process
Plan knowledge (scripts) Problem domain knowledge Intermediate-level programming concepts
Hypothesized new plan
Mental representation
Higher order plans
Hypothesis/ subgoals
Situation model (final mental representation)
Functional and dataflow view
Xref map

Figure 5. Pennington comprehension model.

cellaneous related documents. Internal representations include plans and schemas.) The understanding process matches external representations to programming plans using rules of discourse to select plans, by setting expectations. Once a match is complete, the internal representation is updated to reflect the newly acquired knowledge. These updated mental representations are subsequently stored as new plans.

Comprehension begins with a high-level goal and then generates detailed subgoals necessary to achieve the higher level goals. Program documentation and code invoke implementation, strategic, or tactical plans, depending on the current mental representation's focus.

Pennington model: bottom-up comprehension

Comprehension develops two different mental representations: a program model and a situation model. The program model is usually developed before the situation model.

PROGRAM MODEL. Pennington[1,11] found that when code is completely new to programmers, the first mental representation they build is a control-flow program abstraction called the program model. This representation, built from the bottom up via beacons, identifies elementary blocks of code control primes in the program.

Pennington uses text structure and programming plan knowledge to explain program model development. The program model is created via the chunking of microstructures into macrostructures and via cross-referencing. Programming plan knowledge, consisting of programming concepts, exploits existing knowledge during the understanding task and infers new plans for storage in long-term memory.

Least recently used (LRU) page replacement for memory management is an example of plan knowledge from the operating systems domain. Data structure knowledge may contain the implementation of a first-in, first-out (FIFO) queue.

SITUATION MODEL. This model also is built from the bottom up, as the program model creates a dataflow/functional abstraction. The model requires knowledge of real-world domains, such as generic operating system structure and functionality for the operating system domain. The situation model is complete once the program goal is reached.

Domain plan knowledge is used to mentally represent the code in terms of real-world objects, organized as a functional hierarchy. For example, the situation model describes the actual code "pcboards = pcboards - sold" as "reducing the inventory by the number of PC boards sold." Lower order plan knowledge can be chunked into higher order plan knowledge.

A situation model is built by cross-referencing and chunking. The matching process takes information from

Table 3. Model evaluation criteria.

Model criterion	Abstraction level	Letovsky	Shneiderman and Mayer	Brooks	Soloway, Adelson, and Ehrlich	Pennington	Integrated model
Static: Knowledge structures	Low	Knowledge base	Syntactic knowledge	Programming domain	Implementation plans	Text-structure knowledge	Program model structures
	Intermediate	Knowledge base	Semantic knowledge	Intermediate domain	Tactical plans	Plan knowledge	Situation structures
	High	Knowledge base	Semantic knowledge	Problem domain	Strategic plans		Top-down structures
Static: Mental representations	Low	Implementation layer	Working memory: Low-level concepts	Hypotheses and subgoals	Plans/schemas	Program model	Program model
	Intermediate			Hypotheses and subgoals	Plans/schemas	Situation model	Situation model
	High	Specification layer	Working memory: High-level concepts	Hypotheses and subgoals	Plans/schemas		Top-down model
Dynamic: Processes	Direction	Top-down or bottom-up	Top-down or bottom-up	Top-down	Top-down	Bottom-up	Top-down or bottom-up
Experimentation	Type	Small-scale code experiments	Small-scale code experiments	Self-observation experiments	Small-scale code experiments	Small-scale code experiments	Large-scale code experiments

the program model and builds hypothesized higher order plans. These new plans are stored in long-term memory and chunked to create additional higher order plans.

Figure 5 graphically represents Pennington's model. The right half illustrates program model building, while the left half describes situation model construction. Text-structure knowledge (control primes; program structure; syntax; programming conventions; and control sequence knowledge such as sequences, iterations, or conditions) and external representations (code, design documents, and so on) are inputs to the comprehension process. Beacons can invoke a particular schema (for example, a swap operation causes the programmer to recall sorting functions). Code statements and their interrelationships form the microstructure. Microstructures are chunked into macrostructures (chunked lines of text organized by control primes). These chunks are stored in long-term memory and used to build even larger chunks.

The program model can change after situation model construction begins. A cross-reference map allows direct mapping from procedural, statement-level representations to a functional, abstract program view. Higher order plans can cause a programmer to reconsider the program model and alter or enhance it as necessary. The programmer may directly modify the text base or use these plans as input to the program model comprehension process.

Integrated Metamodel

The integrated code comprehension model (see Figure 6),[5] has four major components: the top-down, situation, and program models and the knowledge base. The first three reflect comprehension processes. The fourth is needed to successfully build the other three. Each component is involved in the internal representation of the program (or short-term memory) and a strategy to build this internal representation. The knowledge base furnishes the process with information related to the comprehension task and stores any new and inferred knowledge. For large systems, a combination of approaches to understanding becomes necessary. Therefore, the integrated model combines the top-down understanding of Soloway, Adelson, and Ehrlich[3] with the bottom-up understanding of Pennington.[1] Nevertheless, experiments show that programmers switch between all three comprehension models.[5]

Any of the three submodels may become active at any time during the comprehension process. For example, during program model construction, a programmer might recognize a beacon 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 programmer then generates subgoals and searches the code for clues to support these subgoals. If the programmer finds a section of unrecognized code during this search, he may return to program model building. Structures built by any one model component are accessible by the other two, but each model component has its own preferred knowledge types.

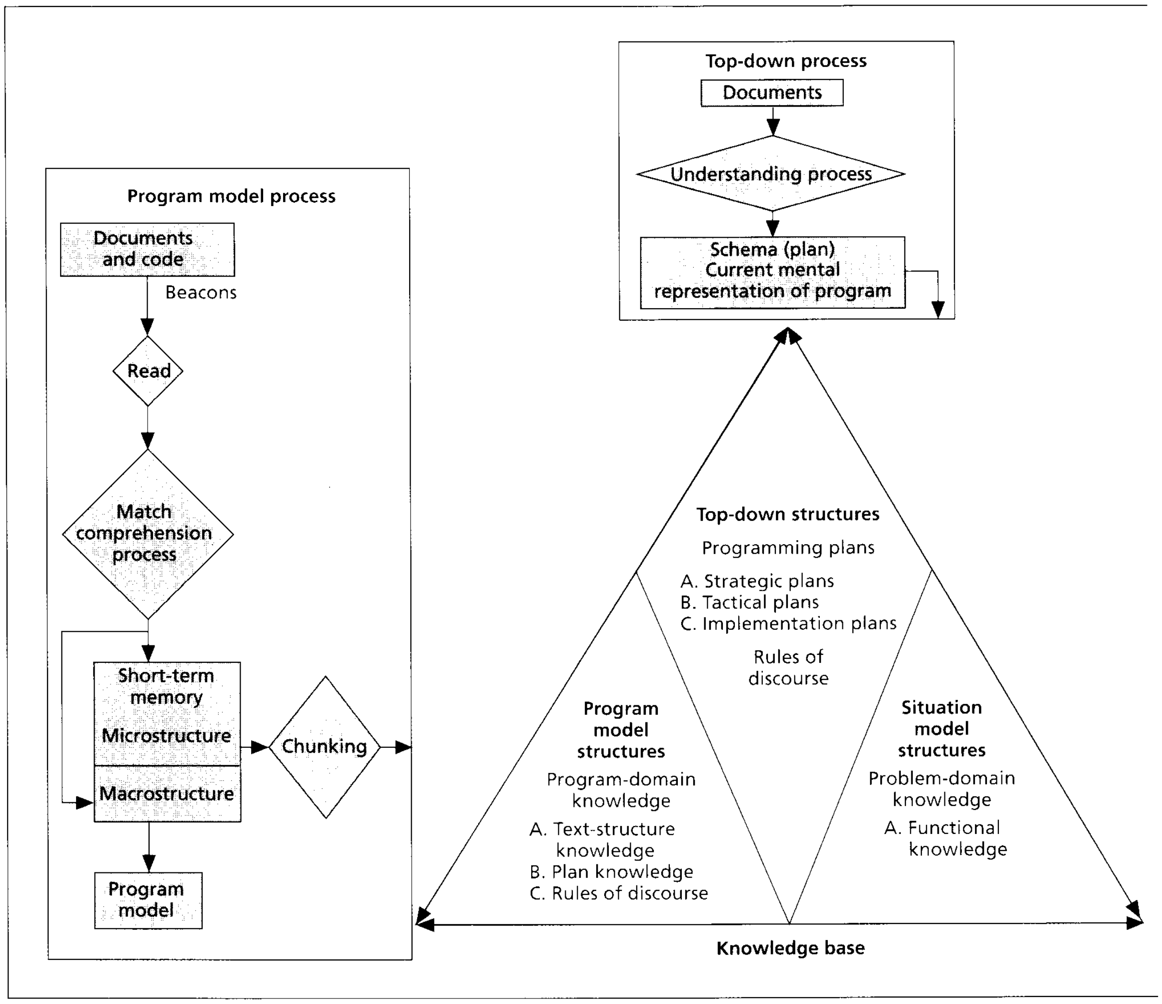


Figure 6. Integrated Metamodel.

EVALUATION AND ANALYSIS

Model scope

Model evaluation uses the following criteria: First, does the model incorporate static structures that represent persistent knowledge and the system's current mental representation? Second, does it represent dynamic processes that build the mental representation using knowledge? Third, to which extent was each model validated by experiments?

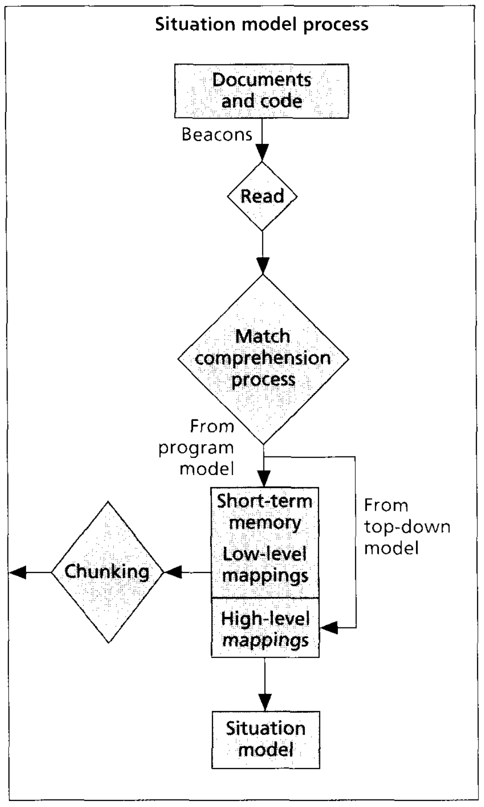
For each model, Table 3 (on the previous page) lists the criteria, the abstraction level for each static property, the direction of dynamic model building, and the experiment type used in validation. Blank cells indicate that the model does not include the associated attribute. For example, Letovsky's model does not have a representation for an intermediate mental representation of the system.

Evaluation

All six models accommodate a mental representation of the code, a body of knowledge (knowledge base) stored in long-term memory, and a process for combining the knowledge in long-term memory with new external information (such as code) into a mental representation. Each differs in the amount of detail for these three main components.

Letovsky's model is the highest level cognition model, emphasizing the mental representation's form. No details explain how the knowledge assimilation process works or how knowledge is incorporated into the mental representation. The knowledge types coincide with Soloway, Adelson, and Ehrlich's model. Shneiderman and Mayer's model is more detailed, since it organizes knowledge in a hierarchy and separates semantic and syntactic knowledge. Like the Letovsky model, this model focuses on the mental representation's form but lacks details on knowledge construction.

Brooks' model differs from the other models in that all changes to the current mental representation stem from hypothesis. The mental model is constructed in one direction only, from the problem domain to the program domain. However, we have observed that program understanding is not unidirectional.[5] Nevertheless, both the Brooks model and the Soloway, Adelson, and Ehrlich model construct the mental representation top-down,



from the highest abstraction level through finer levels of detail. The Brooks model leaves the knowledge structures undefined. Although hypotheses are important cognition drivers, mental representations can be updated by other means—for example, a strategy-driven method (using a cross-referencing, systematic, or opportunistic strategy).

Pennington's model is more detailed and includes specific descriptions of the cognition processes and knowledge. It accounts for the types, forms, and composition of knowledge to construct most of the mental representation. It also contains mechanisms for abstraction. The major drawback is its lack of higher level knowledge structures such as design or application-domain knowledge.

Soloway, Adelson, and Ehrlich's model emphasizes the highest level abstractions in the mental model. One aspect that distinguishes this model is its top-down development of the mental model with its assumption that the knowledge it uses has been previously acquired. By itself, this model does not account for situations when code is novel and the programmer has no experience to use as a backplane in which to plug new code.

Each model represents important aspects of code comprehension, and many characteristics are shared among different models. For example, Brooks, Letovsky, and Shneiderman and Mayer all focus on hierarchical layers in the mental representations. Brooks and Soloway, Adelson, and Ehrlich use a form of top-down program comprehension, while Pennington uses a bottom-up approach to code understanding. And Letovsky and Shneiderman and Mayer use both top-down and bottom-up comprehension. All five models use a matching process between what is already known (knowledge structures) and the artifact under study. No one model accounts for all behavior as programmers understand unfamiliar code. The Integrated Metamodel responds to the cognition needs for large software systems. It combines relevant portions of the other models and adds behaviors not found in them—for example, when a programmer switches between top-down and bottom-up code comprehension.

Analysis

Cognition models and the theories behind them must be grounded in experiments that either collected the information supporting those theories or validated it.

Several types of program comprehension experiments have been conducted. The appropriate type depends on the purpose of the study and the amount of existing research. Objectives range from theory building to validation of detailed portions of a theory. Studies to build a theory are typically observational. Behaviors are observed as they occur in the real world. Once a theory is built from observations, correlational studies can be designed. These studies determine enough about the behavior in question to explore relationships among variables. At the other end of the spectrum, hypothesis-testing experiments investigate cause and effect between variables. Hypothesis testing consists of carefully controlled experiments whose purpose is to validate an existing theory. Correlational studies indicate possible relationships between variables.

Let us now analyze what aspects of code cognition have been covered by existing experiments and how much we really know about how programmers understand code. Table 4 organizes code cognition experiments in terms of models, common cognition elements, code size, programming language, and subjects. For code size, "small" refers to programs of less than 900 lines of code (LOC); "medium" refers to between 900 and 40,000 LOC; while "large" refers to more than 40,000 LOC. Languages such as Cobol, Fortran, Basic, and Pascal are distinguished from state-of-the-art development environments such as C/Unix with tools like Make or Lint. Subjects are categorized as novices, grad students, or professional programmers. Each cell in the table represents the number and types of experiments that investigated the row component with the column attribute. Experiments are classified as observational (O), correlational (C), or hypothesis-testing (H). For more details on these experiments, see von Mayrhauser and Vans.[12]

Many experiments validate or verify portions of the models presented here. For example, several experiments have addressed the use of plans and strategies. Results seem to support the top-down theory of program comprehension. On the other hand, only observational exper-

Table 4. Number and type of experiments—observational (O), correlational (C), or hypothesis-testing (H)—that investigated each component with a given code cognition attribute.

	Code size			Language		Subject		
Component	Small	Medium	Large	Pascal, Fortran, Cobol	C, environments	Novice	Grad student	Professional
Top-down			O:1		O:1			O:2
Situation	O:1		O:1	O:1	O:1			O:2
Program	O:1		O:1	O:1	O:1			O:2
Other	O:2			O:3		O:1		O:3
Processes			O:1		O:1		O:1	
Hypotheses	O:2			O:2		O:1		O:2
Strategies	O:5			O:5		O:1	O:2	O:4
	C:1			C:1				C:1
	H:1			H:1				H:1
Plans	O:1			O:1		O:1	C:1	O:1
	C:1			C:1		C:1	H:2	H:1
	H:3			H:3		H:2		
Rules of discourse	H:1			H:1		H:1	H:1	
Chunks	O:1			O:1		C:1		O:1
	C:1			C:1				C:1
	H:1			H:1				H:1
Episodes	O:1		O:1	O:1	O:1	O:1		O:2
Beacons	H:2	O:1		O:1	O:1	H:1	O:1	H:1
		H:2		H:3	H:2		H:3	
Text structures	O:1			O:1				O:1
Actions	O:1		O:1	O:1	O:1			O:2

iments have examined the use of text structures during bottom-up processing. Correlational and hypothesis-testing experiments are needed to lend credence to the situation models and program models of program comprehension.

Table 4 clearly reveals the unexplored areas of code comprehension. Only one experiment investigating cognition elements used large-scale code. Just two experiments used medium-sized code. A C/Unix environment, probably the most commonly used state-of-the art environment today, appeared in only three experiments. As far as purpose, most studies investigated strategies, beacons, or plans. Few studies addressed processes, rules of discourse, actions, program comprehension episodes, or entire cognition models. In addition, most were observational experiments—which is appropriate, since the development of program-comprehension models is still in the theory-building phase. In light of this information, program cognition must concentrate on three issues: scalability of experiments, static or dynamic behavior, and theory building.

SCALABILITY OF EXPERIMENTS. We must investigate whether the many well designed experiments that use small-scale code can scale up for production code. For example, Pennington's study of mental representations of code[1] used 200-LOC programs. These studies can answer questions about understanding small program segments or very small programs. However, they do not address the interactions of isolated components of understanding or whether these results can play an important role in understanding large programs.

STATIC OR DYNAMIC BEHAVIOR. Current results mainly focus on the static properties of programming skills. For example, experiments identified persistent knowledge areas, such as searching or sorting algorithms, but did not investigate knowledge use or application.

THEORY BUILDING. Many experiments are designed to measure specific conditions (for example, whether or not programmers use plans) but the experimental hypotheses (for example, that programmers use plans when understanding code they have never seen before) are not always based on a well-defined program-comprehension theory. Theories regarding large-scale program comprehension for specialized maintenance tasks are in their infancy.

PROGRAM UNDERSTANDING IS A KEY FACTOR in software maintenance and evolution. We have summarized the major elements of cognition, how they are represented in several different program models, and the importance of experimentation in developing and validating model theories. Several conclusions can be drawn from this survey:

- While a lot of important work exists, most of it centers around general understanding and small-scale code.
- Existing results from related areas need further investigation. For example, Pennington's model borrows from work in understanding technical reports. Perception and problem solving are also strongly related to program comprehension.

- Some results generalize or appear as components of larger results. For example, elements of Pennington's and Soloway, Adelson, and Ehrlich's models appear in the Integrated Metamodel.
- Data availability is a challenge. Finding expert software engineers that work on production code is difficult unless the companies that maintain large-scale code encourage their maintenance engineers to participate in program comprehension experiments. Existing experiments inadequately address these work situations.
- The literature fails to provide a clear picture of comprehension processes based on specialized maintenance tasks such as adaptive or perfective maintenance. Models of the general understanding process play an important part in developing a complete understanding of a piece of code. However, they may not always be representative for narrow tasks such as reuse or enhancements that might more efficiently employ strategies geared towards partial understanding.

A better grasp of how programmers understand code and what is most efficient and effective can spawn various improvements, such as better tools, better maintenance guidelines and processes, and documentation that supports the cognitive process. ∎

References

1. N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, Vol. 19, 1987, pp. 295-341.
2. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Eng.*, Vol. SE-10, No. 5, Sept. 1984, pp. 595-609.
3. E. Soloway, B. Adelson, and K. Ehrlich, "Knowledge and Processes in the Comprehension of Computer Programs," in *The Nature of Expertise*, M. Chi, R. Glaser, and M. Farr, eds., A. Lawrence Erlbaum Associates, Hillsdale, N.J., 1988, pp. 129-152.
4. R.S. Rist, "Plans in Programming: Definition, Demonstration, and Development," *Proc. First Workshop Empirical Studies of Programmers*, Ablex Publishing, Norwood, N.J., 1986, pp. 28-47.
5. A. von Mayrhauser and A.M. Vans, "Comprehension Processes During Large-Scale Maintenance," *Proc. 16th Int'l Conf. Software Engineering*, IEEE CS Press, Los Alamitos, Calif., Order No. 5855, 1994, pp. 39-48.
6. S. Letovsky, "Cognitive Processes in Program Comprehension," *Proc. First Workshop Empirical Studies of Programmers*, Ablex Publishing, Norwood, N.J., 1986, pp. 58-79.
7. I. Vessey, "Expertise in Debugging Computer Programs: A Process Analysis," *Int'l J. Man-Machine Studies*, Vol. 23, 1985, pp. 459-494.
8. R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l J. Man-Machine Studies*, Vol. 18, 1983, pp. 543-554.
9. B. Shneiderman and R. Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," *Int'l J. Computer and Information Sciences*, Vol. 8, No. 3, 1979, pp. 219-238.
10. R. Guindon, "Knowledge Exploited by Experts during Software Systems Design," *Int'l J. Man-Machine Studies*, Vol. 33, 1990, pp. 279-182.
11. N. Pennington, "Comprehension Strategies in Programming," *Proc. Second Workshop Empirical Studies of Programmers*, Ablex Publishing, Norwood, N.J., 1987, pp. 100-112.
12. A. von Mayrhauser and A.M. Vans, "Program Understanding: A Survey," Tech. Report CS-94-120, Colorado State University, Fort Collins, Col., 1994.

***Anneliese von Mayrhauser** is a professor of computer science at Colorado State University. She is also the director of the Colorado Advanced Software Institute, an organization of the state of Colorado for technology transfer research. Her research interests focus on software engineering, particularly testing and maintenance. Von Mayrhauser received a Dipl. Inf. degree from the University of Karlsruhe and an AM degree and PhD from Duke University, all in computer science. She is the author of a software engineering text and over 80 journals and conference publications. She is the IEEE Computer Society's vice president for publications.*

***A. Marie Vans** is a PhD candidate at Colorado State University. As a software research engineer for Hewlett-Packard, she developed specifications and tests for computer hardware. Her research interests include empirical studies of programmers and design of experiments for software engineering. Vans received an MS degree from Colorado State University and a BS degree from California State University at Sacramento, both in computer science. She is a member of Upsilon Pi Epsilon.*

Readers can contact the authors at the Department of Computer Science, Colorado State University, Fort Collins, CO 80523; e-mail {avm, vans}@cs.colostate.edu.