Department of Computer Science

Program Understanding – A Survey

A. von Mayrhauser and A. M. Vans

Technical Report CS-94-120

August 23, 1994

Colorado State University

Program Understanding – A Survey

A. von Mayrhauser
A. M. Vans

Department of Computer Science
Fort Collins, CO 80523
email: avm@cs.colostate.edu
vans@cs.colostate.edu

1 Introduction

Program Understanding or Code Cognition is a central activity during software maintenance, evolution, and reuse. Some estimate that up to 50% of the maintenance effort is spent in trying to understand code. Thus a better grasp of how programmers understand code and what is most efficient and effective can lead to a variety of improvements: better tools, better maintenance guidelines and processes, and documentation that supports the cognitive process.

Maintenance, evolution, and reuse encompass many different tasks. Table 1 lists the most common. Each task consists of a variety of activities. Table 1 also lists the cognitive needs for each activity. Since these activities all have their own specific objective, one would assume they also have their own most effective method of understanding code.

For years, researchers have tried to understand how programmers comprehend programs [5, 6], [21], [24, 25], [28], and [31, 32]. Table 2 summarizes these major cognition models by type of maintenance task and type of model. Comparing them to Table 1 shows that not all maintenance tasks have been investigated for their cognitive processes. On the other hand, we find models that describe general understanding. The objective in those is usually to understand all of the code rather than to understand for a particular purpose like debugging. While models of the general understanding process play a very important part in furthering insight into complete understanding of a piece of code, they may not always be on the mark for special tasks which may more efficiently employ strategies geared towards partial understanding. For example, existing literature about general understanding reports that cognitive results are best when code is systematically understood [20]. This requires that *all* code is understood in detail. For large-size software or specific tasks like "find the interface error in X", this does not seem feasible nor desirable. Preferably, one would like specialized cognition processes that, because they have more focus, are more efficient. The next question becomes, (how) does special task cognition differ from general cognition? And, what do all models have in common?

Section 2 describes common elements of cognition models. In all of them existing knowledge is used to build new knowledge about the software (the mental model). People employ various strategies and use cues in code or documentation as guidance. We also know that the level of expertise greatly affects effectiveness and efficiency of code understanding. Section 3 describes the cognition models of Table 2 in more detail. Many of the models are based on exploratory experiments. Some have been validated with subsequent experiments. Thus experimentation plays a major role in the definition and validation of code cognition models. Section 4 describes the experimental paradigm of code cognition experiments. Major phases of all experiments include definition, planning, operation, and interpretation. Section 4 also explains the role of various experimental techniques in code cognition experiments and reports on the experimental design of existing code cognition experiments. This makes it easier to determine whether existing results can be applied to a new situation.

Maintenance Task	*Activities*	*Explanation*
Adaptive	Understand system	Understand existing system
	Define adaptation requirements	Requirements are extended from original requirements
	Develop prelim & detailed adaptation design	More design constraints due to existing system
	Code Changes	Merge adaptation code into existing code
	Debug	Focus on correct behavior of recently added code
	Regression tests	Develop tests for new code, run old tests to ensure other code not affected
Perfective	Understand system	Understand existing system
	Diagnosis & requirements definition for improvements	Identify exact nature of need for improvement e.g. performance improvements
	Develop prelim & detailed perfective design	Requirements are extended from original requirements
	Code changes/additions	Merge perfective code into existing code
	Debug	Focus on correct behavior of recently perfected code
	Regression tests	Develop tests for perfected code, run old tests to ensure other code not affected
Corrective	Understand system	Understand existing system
	Generate/Evaluate hypotheses concerning problem	Hypotheses about nature of problem generated using symbolic code execution, watching system behavior
	Repair code	Rewrite defective code or add omitted code
	Regression tests	Test for system stability after change
Reuse [18]	Understand the problem, find solution based on close fit with predefined components	Reuse code AS IS, Design is open-ended, not initially constrained but defined by plugging some configuration of reusable components
	Obtain predefined components	Find reusable components
	Integrate predefined components	Similar to integrating new code developed by several engineers
Code Leverage	Understand problem, find solution based on predefined components components	Design fixed or constrained by problem, design exists & close to desired solution, may use new code
	Reconfigure solution to increase likelihood of using predefined components	Find all possible solutions, use solution with highest leverageable code
	Obtain & modify predefined components	Find leverageable components
	Integrate modified components	Similar to integrating new code developed by several engineers

Table 1: Tasks and activities requiring code understanding

Section 5 discusses important open issues in program understanding, particularly considering maintenance and evolution of large scale code. They relate to scalability of existing results with small programs, validity and credibility of results based on experimental procedure, and challenges of availability of data. Section 6 summarizes the current status of program comprehension results as they apply to relevant programming tasks during maintenance and evolution.

2 Common Elements of Cognition Models

Program comprehension is a process that uses *existing knowledge* to acquire *new knowledge* that ultimately meets the goals of a code cognition task. In the process we reference both existing and newly acquired knowledge to build a *mental model* of how the software works. How we go about understanding depends on *strategies*. While cognition strategies vary, they share the process of formulating *hypotheses* and then resolving, revising, or abandoning them.

Knowledge

Programmers possess two types of knowledge, general knowledge that is independent of the specific software application they are trying to understand, and software specific knowledge that represents their current level of understanding of the software application. During the understanding process they acquire more software specific knowledge, but may also need more general knowledge (e. g. how a round-robin algorithm works). Existing knowledge includes knowledge of the programming languages and the computing environment, programming principles, application domain specific architecture choices, algorithms, and possible solution approaches. When the programmer has worked with the code before and knows something about it, existing knowledge includes any (partial) mental model of the software.

New knowledge is primarily knowledge about the software product. It is acquired throughout the code understanding process as the mental model is built. This knowledge relates to functionality, software architecture, how algorithms and objects are implemented, control, and data flow,etc. Obviously it spans a variety of levels of abstraction from "this is an operating system" to "variable q is incremented in this loop".

The understanding process matches existing knowledge with software knowledge until there are enough matches to satisfy the programmer that he understands the code. The set of matches is the mental model. It can be complete or incomplete.

Mental Model

The mental model is a current internal (working) representation of the software under consideration. It contains various *static* entities including *text structures*, *chunks*, *plans*, and *hypotheses*. The mental model can be described in terms of a hierarchy of plans, chunks, and text structures. 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. It may be constructed using a combination of several *dynamic* behaviors including *strategies*, *actions*, and *processes*.

Static Elements of the Mental Model

Text Structure knowledge includes the program text and its structure. Text-structure knowledge [19] for understanding is built through experience and is stored in long-term memory. Pennington [24] uses text-structure knowledge to explain control-flow knowledge for program understanding. Structured programming units form text structure and are the knowledge organization in her comprehension model.

Examples of text structure knowledge units are: *Control-Primes* – Iteration (loop constructs), sequence, and conditional constructs (eg. if-then-else); *Variable definitions*; *Module calling hierarchy*; and *Module parameter definitions.* This *Micro-structure* of the program text consists of the actual program statements and their relationships. For example, the statement **BEGIN** signifies the start of a block of code, while a subsequent **IF** indicates a conditional control structure for some purpose. The relationship between these two propositions is that the **IF** is part of the block initiated by the **BEGIN**.

Chunks are knowledge structures consisting of various levels of abstractions of text structures. Text-structure chunks are called *Macro-structures* which are identified by a label and correspond to control-flow organization of the program text [24]. For example, the micro-structure for a sort consists of all its statements. The macro-structure is an abstraction of the block of code and consists only of the label *sort.* Lower level chunks can form higher level chunks. Higher level chunks consist of several labels and the control-flow relationships between them.

Plans are elements of knowledge that support the development and validation of expectations, interpretations, inferencing, and keep the attention of the comprehender during the program understanding task. These plans also include causal knowledge about the information flow and relationships between parts of programs. Plans are schemas or *frames* 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.

Programming plans can be high-level, low level, or intermediate level programming concepts. For example, searching, sorting, and summing algorithms as well as data-structure knowledge including arrays, linked-lists, trees, and stacks are intermediate level. Iteration and conditional code segments are low-level concepts that can be plans or components of plans as a chunk of text structure. 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. Useful plans when developing a software tool for designing automobiles include schemas related to the function and appearance of a generic car. Slots for problem domain objects such as steering wheels, engines, doors, and tires are necessary components of an appropriate plan. For program understanding, these 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 are also concerned with the environment surrounding the software application, the domain specific architecture and solution alternatives.

Letovsky [21] refers to *Hypotheses* as conjectures and defines them as comprehension activities (actions) that take on the order of seconds or minutes to occur. Letovsky identified three major types of hypotheses: *Why* conjectures which hypothesize the purpose of some function or design choice, *How* conjectures hypothesize about the method for accomplishing a program goal, and *What* conjectures which hypothesize about what something is, for example a variable or function. Additionally, there are degrees of certainty associated with a conjecture and these vary from uncertain guesses to almost certain conclusions.

Brooks [6] theorizes that hypotheses are the only drivers of cognition. This theory states that understanding is complete when the mental model consists entirely of a complete hierarchy of hypotheses. At the top of this hierarchy is the *primary hypothesis* which is a high-level description of the program structure. It is necessarily global and non-specific. Once the primary hypothesis is generated, subsidiary hypotheses that support the primary hypothesis are generated. This process is continued until the mental model is built. Brooks also considers three reasons hypotheses sometimes fail: code to verify a hypothesis can't be found; confusion due to a single piece of code that satisfies different hypotheses; and code that can not be explained.

Hypotheses are important drivers of cognition. They help to define 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.

In short, the mental model is one plan composed of many subplans representing different levels of abstraction. Each plan represents software specific or software independent information with slots and fillers for other plans or chunks of text structure.

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 may be to go about it *systematically* by reading and understanding every single line of code while building a mental representation at higher and higher levels of abstraction. An *opportunistic strategy* studies code in a more haphazard fashion. Littman et al [22] found that programmers who used a systematic approach to comprehension were more successful at modifying code (once they understood it) than programmers who took the opportunistic approach. On the other hand, for large programs systematic understanding may not be possible.

Strategies also differ in how to match programming plans to code. *Shallow reasoning* [31, 32] does so without in-dept analysis. Many experts do this when they recognize familiar plans. *Deep reasoning* [31, 32] looks for causal relationships among procedures or objects and performs detailed analyses.

Strategies guide understanding mechanisms that produce information. Two such mechanisms are *chunking* and *cross-referencing. Chunking* creates new higher-level-abstraction structures from *chunks* of lower-level structures. As groups of 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 level of abstraction. For example, a piece of code may represent a linked-list definition as pointers and data. In an operating system definition this may be abstracted as a "ready-queue".The section of code that takes a job from the ready queue, puts it into the running state, monitors elapsed time, and removes the job after the time quantum has expired may be abstracted as a "round-robin scheduler". The fragments of code for the queue, the timer, and the scheduling are micro-structures. Continued abstraction of 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 operation system".

Cross-referencing relates different levels of abstraction, e. g. a control-flow view and a functional view by mapping program parts to functional descriptions. For instance, once we know that a segment of code performs process management and know why the code exists, we have made a statement about functionality. Cross-referencing is thus an integral part of building a complete mental representation across all levels of abstraction.

If we look at code cognition as a process that formulates hypotheses and then checks whether they are true or false and revises them where necessary, then hypotheses are programmer-defined goals. Programmers are trying to match these goals. Goals exist at all levels of abstraction, like plans and schemas. The essence of an effective and efficient strategy is to keep the number of open hypotheses manageable while increasing understanding incrementally.

Actions classify programmer activities, both implicit and explicit during a specific maintenance task. Examples of action types include "asking a question" and "generating a hypothesis". Actions are important because they define *episodes*. Episodes are composed of sequences of actions. Then, episodes aggregate to form higher-level *Processes*. Thus, processes, episodes, actions, and strategies are the dynamic components of mental model construction.

Facilitating Knowledge Acquisition

Beacons are cues that index into knowledge. Beacons can be text or a component of other knowledge. For example, a swap statement inside a loop or a procedure can act as a beacon for a sorting function; so can the procedure name *Sort*. Wiedenbeck [39] used short Pascal programs in a recall experiment designed to study whether programmers actually use beacons during program comprehension activities. Experienced programmers were able to recall beacon lines much better than novices. Beacons are useful for gaining a high level understanding in processes such as top-down comprehension.

Gellenbeck & Cook [14] investigated the role of mnemonic procedure and variable names as beacons in understanding code. While the study confirmed the usefulness of beacons in general, no useful conclusions regarding the strength of variable names versus procedure names could be shown.

Rules of discourse are conventions in programming, similar to dialogue rules in conversation. Examples are coding standards, common forms of algorithm implementations, expected use of data structures, mnemonic naming, etc. Rules of discourse set expectations of programmers. Programming plans are retrieved from long term memory using these expectations. Soloway and Ehrlich [31] showed that rules of discourse had a significant effect on the ability of expert programmers to comprehend code. The experiment required programmers to understand one program that was written using *plan-like* code and a second program that used *unplan-like* code. Plan-like code is defined as code fragments that match expert programming plans. They were able to show that the programmers performed significantly better on the plan-like code than on the unplan-like code. The performance of experts dropped to that of novices when attempting to understand the unplan-like code since they were unable to match this code to any programming plans stored in long-term memory. In practice, this means that unconventional algorithms and programming styles are much harder to understand, even for experts.

Table 3 summarizes the static and dynamic components for the mental model.

Expert Characteristics

The level of expertise in a given domain greatly affects the efficiency and the success of a programmer during program understanding. Experts tend to show the following characteristics:

- Experts organize knowledge structures by functional characteristics of the domain in which they are experts. Knowledge possessed by novices is typically organized by surface features of the problem. For instance, novices may have knowledge about a particular program organized according to the program syntax. An example of a functional category is algorithms. Experts may organize knowledge about programs in terms of the algorithms applied rather than the syntax used to implement the program, [17].

- Experts have efficiently organized specialized schemas developed through experience. A high-level design study conducted by Guindon, [17], indicated that experts not only used general problem solving strategies such as divide-and-conquer, but also more specialized design schemas. These schemas differed in granularity and seemed to be abstracted from previously designed software systems. The schemas had comparable structures, but different problem domains.

- Specialized schemas contribute to efficient problem decomposition and comprehension, [17]. For problems that match specialized schemas, top-down comprehension becomes feasible.
- Vessey [34], conducted several debugging experiments and found that experts are flexible in approaches to problem comprehension. In addition, experts are able to let go of questionable hypotheses and assumptions more easily. Experts tend to generate a breadth-first view of the program and then refine hypotheses as more information becomes available.

The common elements of program cognition models occur in a variety of existing theories. The next section presents the most important of these code cognition models.

Model	*Maintenance Task*	*Reference*
Top-Down	Understand	E.Soloway & K.Ehrlich, **Empirical Studies of Programming Knowledge** In: IEEE Transactions on Software Engineering, Vol.SE-10, No. 5, 1984.
		R.Rist, **Plans in Programming:Definition, Demonstration, and Development** In: Empirical Studies of Programmers, Eds. Soloway & Iyengar, ©1986, Ablex Publishing Corp.
Control-Flow	Understand	N.Pennington, **Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs**, In: Cognitive Psychology, 19, 1987.
Functional	Understand	N.Pennington, **Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs**, In: Cognitive Psychology, 19, 1987.
Integrated	Understand, Corrective, Adaptation, & Perfective	A.von Mayrhauser & A.Vans, **From Program Comprehension to Tool** *Requirements for an Industrial Environment*, In: Proceedings of the 2nd Workshop on Program Comprehension, Capri, Italy, July 1993, pp. 78 -86.
Other	Enhancement	S.Letovsky,**Cognitive Processes in Program Comprehension**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation
	Corrective	Iris Vessey, **Expertise in debugging computer programs:A process analysis**, In: International Journal of Man-Machine Studies, (1985)23.
	Understand	R.Brooks,**Towards a theory of the comprehension of computer programs**, In: International Journal of Man-Machine Studies, 18(1983).
		B.Shneiderman,**Exploratory Experiments in Programmer Behavior**, In: International Journal of Computer and Information Sciences, Vol. 5, No.2, 1976

Table 2: Code Cognition Models

Dynamic Behaviors	*Static Entities*
Strategies	Text Structure
Actions	Chunks
Episodes	Plans
Processes	Hypotheses
	Beacons
	Rules of Discourse

Table 3: Common Elements of Cognition

3 Cognition Models

3.1 Letovsky Model

Letovsky's comprehension model, [21], has three main components – a knowledge base, a mental model (internal representation), and an *assimilation process* . This model is a very high level cognitive model of program understanding. The knowledge base consists of programming expertise, problem domain knowledge, rules of discourse, plans (similar to Pennington's text-structure knowledge and plan knowledge), and goals.

The mental model consists of three layers – a specification, an implementation, and an annotation layer. The specification layer contains a complete characterization of the program goals. This is also the highest level of abstraction of the program. The implementation layer contains the lowest level abstraction with data structures and functions as entities. The annotation layer ties each goal in the specification layer to it's realization in the implementation layer. Understanding and thus the links between specification layer and implementation layer can be incomplete. The *dangling purpose unit* models such unresolved links.

The assimilation process can occur in either a top-down or bottom-up fashion. It is opportunistic in that the understander proceeds in a way she feels yields the highest return in the form of knowledge gain. Understanding proceeds by matching code, documents, etc. with elements from the knowledge base with the sole purpose of contributing to one of the three layers constructed in the mental representation. Figure 1 represents Letovsky's model.

3.2 Shneiderman Model

The Shneiderman comprehension model is shown in figure 2 [28]. Program comprehension involves recoding the program in short-term memory via a chunking process into an *internal semantic representation* using working memory. These internal semantics consist of different levels of abstraction of the program. At the top are high-level concepts like program goals. At the lowest levels are details such as the algorithms used to achieve program goals.

Long-term memory helps during internal semantics construction. Long-term memory is a knowledge base with semantic and syntactic knowledge. Syntactic knowledge is programming language dependent while semantic knowledge consists of general programming knowledge independent of any specific programming language. Like working memory, semantic knowledge in long-term memory is multi-leveled and incorporates high-level concepts and low-level details. Design works forward from the problem statement to the program while program understanding starts with the program and works to the problem statement.

3.3 Brooks Model

Brooks, [6], defines program comprehension as the reconstruction of the domain knowledge used by the initial developer. Domain knowledge is knowledge about a particular domain such as operating systems or UNIX systems. In this theory, understanding proceeds by recreating the mappings from the problem domain through several intermediate domains into the programming domain. The problem domain or *application domain* consists of problems in the real world.

An example of a problem in the application domain might be the maintaining of appropriate levels of inventory in order to keep back-orders to a minimum and at the same time minimizing exposure to loss due to obsolete inventory. The objects are inventories whose levels must be closely monitored to meet the constraints of the problem. These are physical entities that have properties such as size, cost, and quantity. In order to construct a program to solve this problem, these objects and their properties must be encoded for use by a computer. Once the physical objects are characterized, intermediate knowledge domains are required. The inventory can be assigned part numbers. Perhaps cost is determined not only by actual cost but also overhead like storage. We need knowledge of accounting practices to recognize the appropriate overhead calculations. Once the equations are identified, we need knowledge of program syntax to implement the equations in a programming language. This example used at least four different knowledge domains to reach the programming domain: inventories, accounting, mathematics, and programming languages.

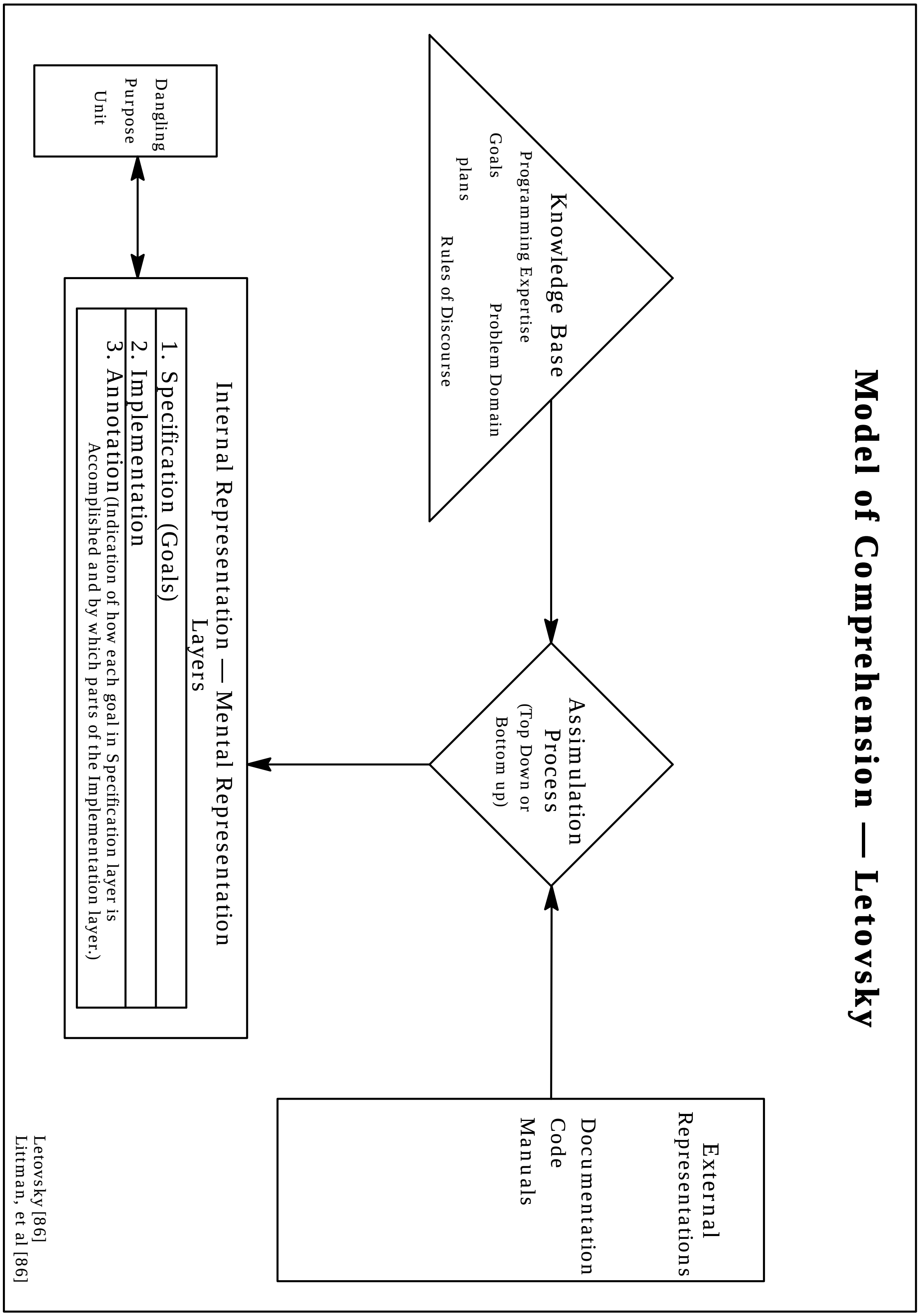

Figure 1: Letovsky – Comprehension Model

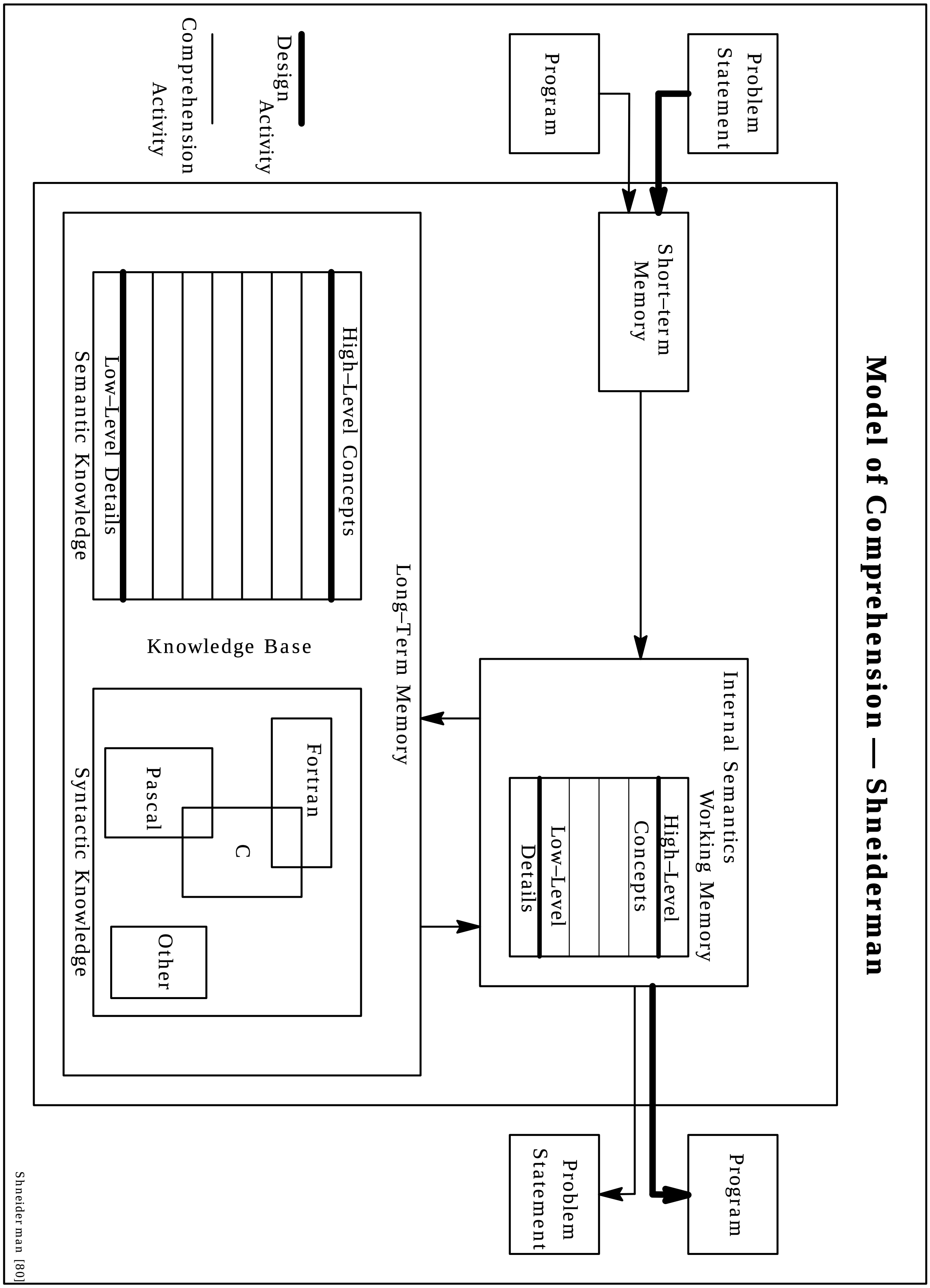

Figure 2: Shneiderman – Comprehension Model

Knowledge within each domain consists of details about the objects in the domain, the set of operations allowed on the objects, and the order in which the operations are allowed. There is also inter-domain knowledge that describes the relationships between objects in different, but closely related domains such as operating systems in general and UNIX in particular.

The mental model is built through a top-down process that successively refines hypotheses and auxiliary hypotheses. Hypotheses pertain to specific domains or connections between knowledge domains. For instance, an hypothesis may state that a particular equation (math domain) expresses cost (accounting domain). Hypotheses can be generated through the recognition of beacons. For example, a procedure name *FCFS* may generate the hypothesis that a first-come-first-serve algorithm is used for process scheduling. Hypothesis generation drives domain knowledge retrieval. Hypotheses are iteratively refined, passing through several knowledge domains, until they can be matched to specific code in the program or some related document.

Figure 3 illustrates this model. Knowledge, shown as triangles, can be used directly for hypothesis generation in the mental model or it can be matched (mapped) from one domain into another. Another 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 and can be used to look at either the internal or external representations for expected information. Verification is also hypothesis driven in that once an hypothesis is generated the external (internal) representations can be searched to support the hypothesis.

3.4 Top-Down Model – Soloway & Ehrlich

Top Down program understanding model [31, 32] typically applies when the code or type of code is familiar. Suppose an expert whose specialty is operating systems, is asked to maintain an operating system she has never before seen. As an expert, she can immediately decompose the new system into elements she knows must be implemented in the code: a process manager, a file manager, an I/O manager, and a memory manager. Each of these can be decomposed e.g, process management includes interprocess communication and process scheduling. Process scheduling can be implemented through one of several scheduling algorithms: e.g. round robin, shortest job first, or priority scheduling. The programmer may continue in this top down fashion until she recognizes a block of code, in this case, the precise process scheduling algorithm. During understanding, it is not necessary to re-learn this algorithm line by line. Instead, the engineer needs only recognize that the appropriate code exists. Theoretically, new code could be understood entirely in a top down manner if the comprehender had already mastered code that performed the same task and the code was structured in exactly the same way. Figure 4 represents this model.

The model uses three types of plans: *Strategic, Tactical,* and *Implementation* plans. *Strategic Plans* describe a global strategy used in a program or an algorithm and specify actions that are language independent. These are the highest-level plans available during comprehension. An example of a strategic plan is the process state model for operating systems. These plans say nothing about the actual constructs used for implementation and contain no lower level detail. Strategic plans can be further decomposed into language independent tactical plans.

Tactical plans are local strategies for solving a problem. These plans contain language independent specifications of algorithms. For an operating system, tactical plans might include cpu scheduling algorithms for the process state model, such as FCFS (First-Come First-Served), Shortest-Job-First, Priority scheduling, or Round-Robin. These knowledge structures may include abstract data-structures, such as queues to keep track of which process to schedule next. The tactical plans composed of these algorithm descriptions are linked to the *process state model* strategic plan. Tactical plans can not be used directly for understanding specific code since they are not tied to particular languages. Tactical plans can include the abstract data types or objects. For instance, a queue may be used in the FCFS algorithm.

Implementation plans are language dependent and are used to implement tactical plans. These plans contain actual code fragments acquired through experience. A First-Come First-Served function written in C is an example of an implementation plan. A queue for FCFS can be implemented as a linked list or array structure. These represent two different implementation plans for the same tactical plan.

A mental model is constructed during top down comprehension and consists of a hierarchy of goals and plans. Rules of discourse and beacons facilitate decomposition of goals into plans and plans into lower-level plans. Typically, *shallow reasoning* is used to build the connections between the hierarchical components.

Figure 4 shows the model's three major components: 1) The triangles represent knowledge (programming plans or rules of discourse). 2) The diamond represents the understanding process. 3) The rectangles illustrate internal or external representations of the program. Understanding matches external representations to programming plans using rules of discourse for help in selecting 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 proceeds with the generation of detailed sub-goals necessary to achieve the higher level goals. The comprehender draws on previously stored knowledge (plans) and programming rules of discourse in an attempt to satisfy the goals. Program documentation and code serve as the tools for invocation of implementation, strategic, or tactical plans, depending on the focus of the current mental representation. In addition to building the mental representation of the current program, top down comprehension also facilitates the building of new programming plans which are in turn stored in long term memory for future use.

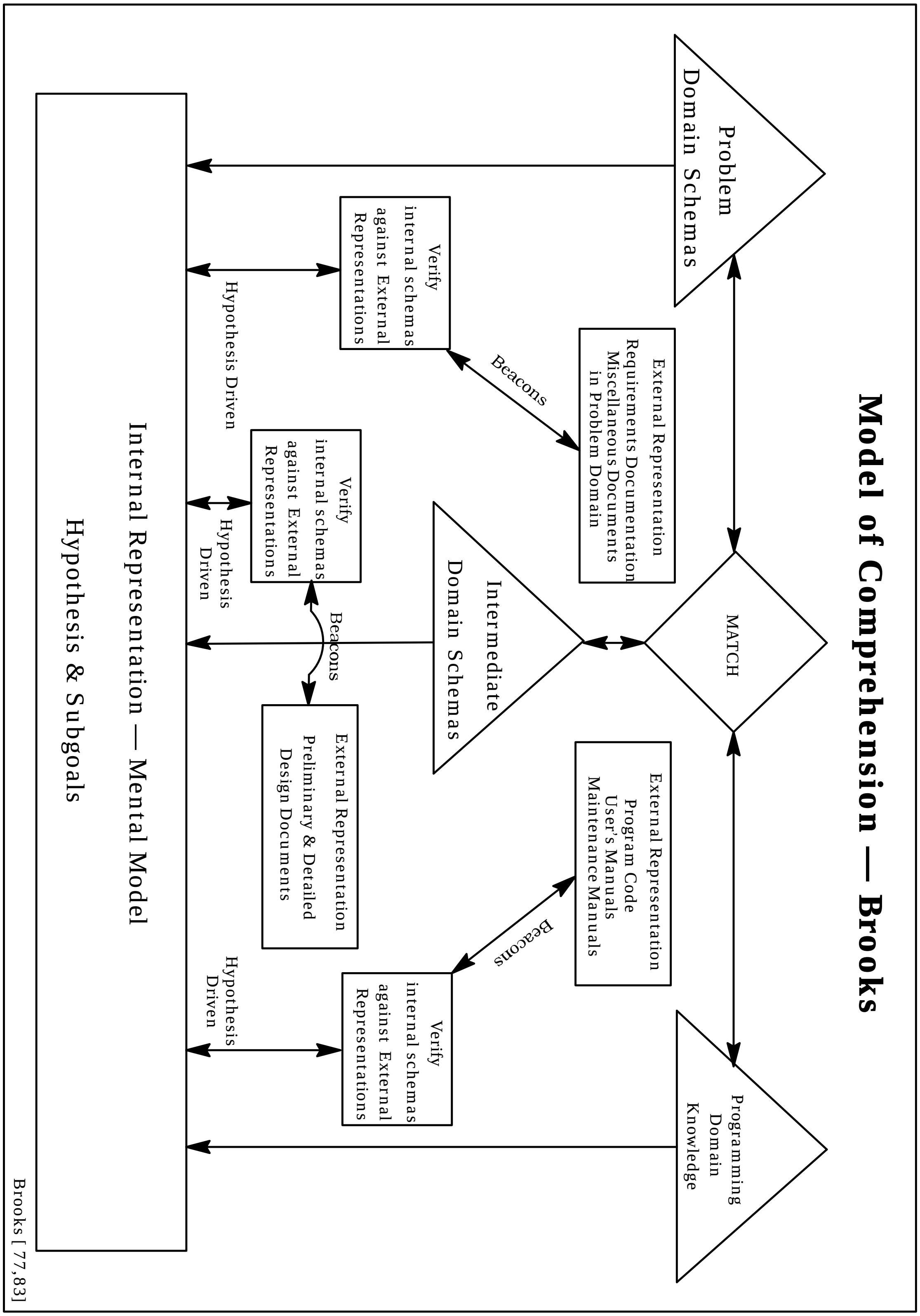
Model of Comprehension — Brooks
Problem Domain Schemas
External Representation Requirements Documentation Miscellaneous Documents in Problem Domain
MATCH
External Representation Program Code User's Manuals Maintenance Manuals
Programming Domain Knowledge
Beacons
Intermediate Domain Schemas
Beacons
Beacons
Verify internal schemas against External Representations
Verify internal schemas against External Representations
External Representation Preliminary & Detailed Design Documents
Verify internal schemas against External Representations
Hypothesis Driven
Hypothesis Driven
Hypothesis Driven
Internal Representation — Mental Model
Hypothesis & Subgoals
Brooks [77,83]

Figure 3: Brooks – Comprehension Model

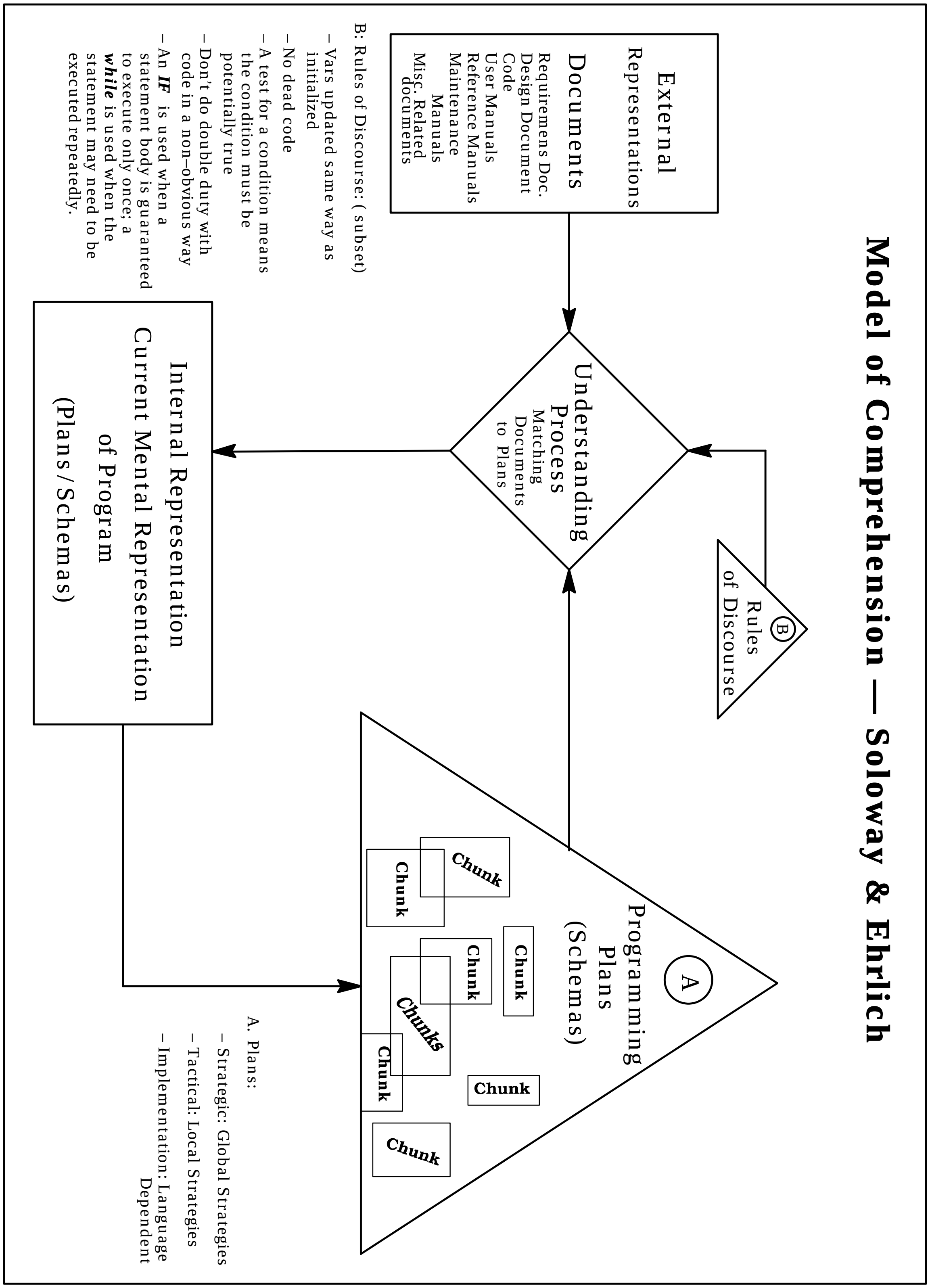
Model of Comprehension — Soloway & Ehrlich
External Representations
Documents
Requiremens Doc.
Design Document
Code
User Manuals
Reference Manuals
Maintenance Manuals
Misc. Related documents
B Rules of Discourse
Understanding Process
Matching Documents to Plans
Internal Representation
Current Mental Representation of Program
(Plans/Schemas)
A Programming Plans (Schemas)
Chunk
Chunk
Chunk
Chunk
Chunks
Chunk
Chunk
Chunk
B: Rules of Discourse: (subset)
– Vars updated same way as initialized
– No dead code
– A test for a condition means the condition must be potentially true
– Don't do double duty with code in a non–obvious way
– 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.
A. Plans:
– Strategic: Global Strategies
– Tactical: Local Strategies
– Implementation: Language Dependent

Figure 4: Soloway & Ehrlich – Comprehension Model

3.5 Pennington's Model – Bottom-Up Comprehension

3.5.1 The Program Model

When code to be understood is completely new to the programmer, Pennington [24, 25] found that the first mental representation programmers build is a control flow abstraction of the program called the *program model*. This representation is built from the *bottom up* using *beacons* to identify elementary blocks of code (*control primes*) in the program. Pennington uses *text structure* and *programming plan knowledge* to explain the development of a program model. This text-structure knowledge consists of the control primes used to build the program model. Programming plan knowledge, consisting of programming concepts, is used to exploit existing knowledge during the understanding task and to infer new plans for storage in long-term memory. Examples of plan knowledge structures from the Operating Systems domain are memory management page replacement algorithms including LRU (Least Recently Used) and NRU (Not Recently Used). Data structure knowledge may contain the implementation of a FIFO queue.

The mental representation is a current internal (working) description of program text and represents current understanding. Two different representations are developed during comprehension – a *Program Model* and a *Situation Model.* The program model is usually developed before the situation model. Text-structure and programming plan knowledge play a critical role in the development of the program model. The program model is created by *chunkingmicro-structures* into *macro-structures* and by cross-referencing.

3.5.2 The Situation Model

Once the program model representation exists, Pennington, [24], 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. Knowledge of real-world domains is required. For the operating systems example this knowledge includes facts about generic operating system structure and functionality. Construction of the situation model is complete once the program goal is reached.

Domain Plan Knowledge is used to derive a mental representation of the code in terms of real-world objects, organized as a functional hierarchy in the problem domain language. For example, the situation model describes the actual code "pcboards = pcboards - sold;" as "reducing the inventory by the number of pc boards sold. This is done to keep an accurate count of inventory". In the same way that the program model consists of a hierarchy of chunked components, the situation model represents chunked plan knowledge. Lower-order plan knowledge can be chunked into higher-order plan knowledge. "The memory manager, the process manager, the secondary storage manager, the I/O system, the file manager, the protection system, networking, and the shell together define the operating system". This is the highest-order plan and it is comprised of lower-order plans containing knowledge about each component.

The mechanisms used for situation model building are the same as those used for program model building: cross-referencing and chunking. The only difference is that the knowledge involved is domain plan knowledge, as opposed to program model text-structure and plan knowledge.

Again, beacons can play an important part in determining which plans are used. The matching process takes information from 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. The situation model as a mental representation contains a functional and data-flow abstraction of the program.

Figure 5 is a graphical representation of Pennington's model. The right half illustrates the process of program model building while the left half describes situation model construction. Text-structure knowledge and any external representations (code, design documents, etc.) are inputs to the comprehension process. Beacons can influence invocation of a particular schema (e.g. a swap operation causes the programmer to recall sorting functions). Code statements and the interrelationships among them are organized into a micro-structure. Micro-structures are chunked into macro-structures. These chunks are stored in long-term memory and subsequently used in the comprehension process to build even larger chunks. Once a control-flow mental representation exists, the program model is established.

Information flows between the program model and the situation model illustrate that the program model

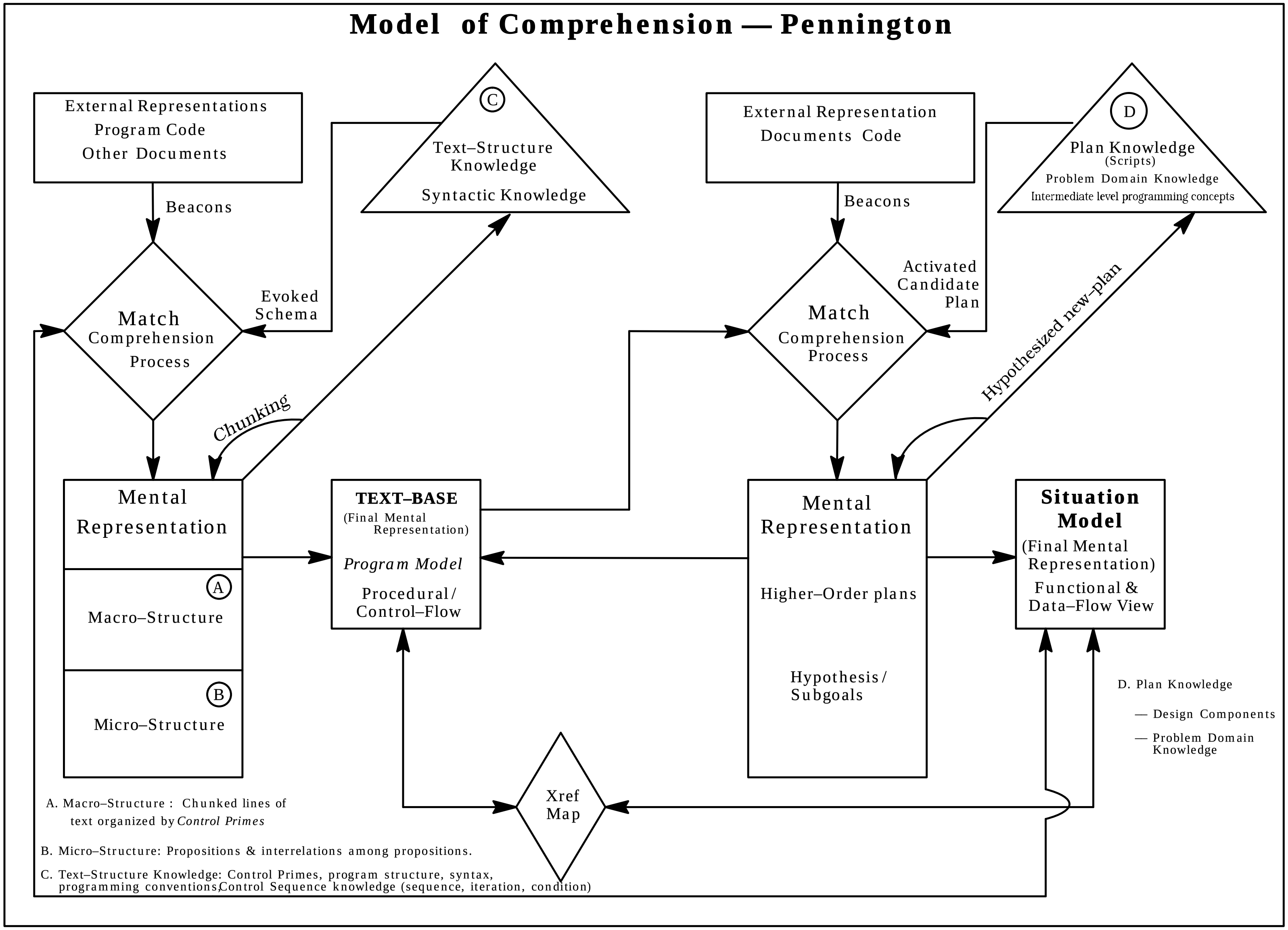

Figure 5: Pennington – Comprehension Model

is modifiable after situation model construction begins. A cross reference map allows a direct mapping from procedural, statement level representations to a functional, abstract view of the program. Higher-order plans can cause a switch to program model building, either directly modifying the text-base or as input to the program model comprehension process.

3.6 Integrated Meta-Model

The integrated code comprehension model [37, 38] consists of four major components, (1.) Top-Down model, (2.) Situation model, (3.) Program model, and (4.) Knowledge base. The first three reflect comprehension processes. The fourth is necessary for successfully building the other three. Each component represents both the internal representation of the program being understood (or short-term memory) as well as a strategy used to build this internal representation. The knowledge base either furnishes the process with information related to the comprehension task or stores any new and inferred knowledge.

The integrated model combines the top-down understanding of [32] with the bottom-up understanding of [24], recognizing that for large systems a combination of approaches to understanding becomes necessary. Experiments showed that programmers switch between all three comprehension models [37, 38].

As Figure 6 illustrates, any of the three sub-models may become active at any time during the comprehension process. For example, during program model construction a programmer may 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 sub-goals (e.g. I need to find out whether the sort is in ascending or descending order) and searches the code for clues to support these sub-goals. 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, each model component has its own preferred types of knowledge.

3.7 Evaluation of Models

At the highest level of generality, all five models accommodate 1) a mental representation of the code, 2) a body of knowledge (knowledge base) stored in long-term memory, and 3) a process for 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 most general cognition model. It focuses on the form of the mental representation . There are no details on how the knowledge assimilation process works or how knowledge is incorporated into the mental representation beyond the statement that it occurs. The types of knowledge coincide with Soloway & Ehrlich's model. Shneiderman's model is more detailed because it includes a hierarchical organization of knowledge and a separation between semantic and syntactic knowledge. Similar to Letovsky, the focus is on the form of the mental representation, but it lacks details on knowledge construction.

Brooks' model is different from the other models in that all changes to the current mental representation occur as the result of a hypothesis. The mental model is constructed in one direction only, from the problem domain to the program domain. The knowledge structures are kept undefined. Although hypotheses are important drivers of cognition, there are other ways of updating the current mental representations, for example strategy-driven (using a cross-referencing strategy or using a systematic or opportunistic strategy). Also, if understanding occurred only from problem to program domain, it would not be possible to switch from one level of abstraction to another going in an opposite direction: suppose an engineer was trying to understand a piece of code she has never seen. This implies she will start building her mental representation from the right-hand side of figure 3 (in the program domain). If at some point she makes a connection to the domain, there is no way to jump back to the left-hand side of the figure (problem domain). At the same

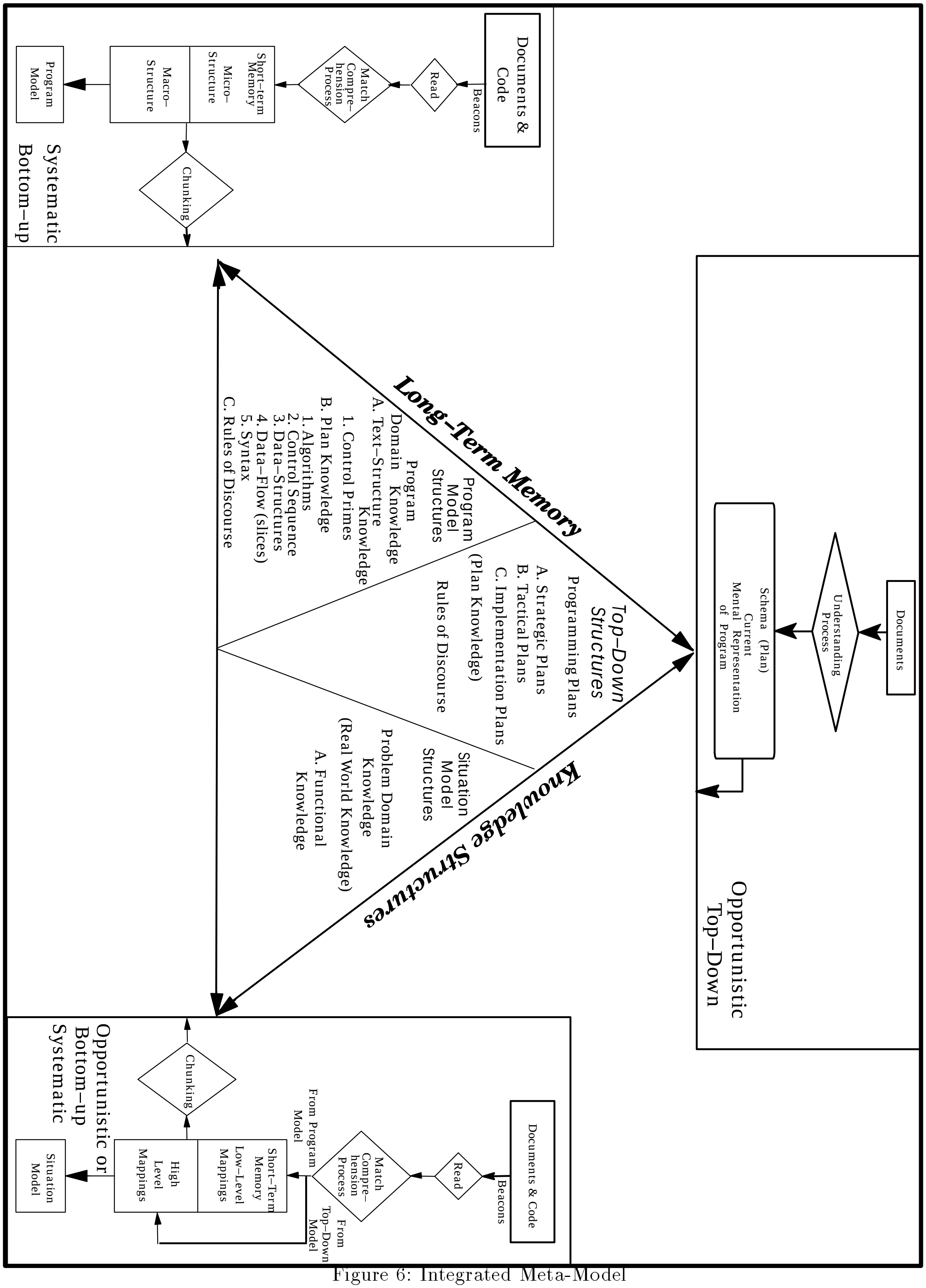

Figure 6: Integrated Meta-Model

time, such cognition behavior has been observed [38]. Brooks' model is similar to the Soloway and Ehrlich model in that the mental representation is constructed top-down through finer levels of details.

Pennington's model is more detailed and includes specific descriptions of the cognition processes and knowledge. It accounts for the types and composition of knowledge to construct most of the mental representation as well as their form. It also contains mechanisms for abstraction. The major drawback of this model is the lack of higher level knowledge structures such as design or application domain knowledge. It is also uni-directional in that comprehension is only built bottom-up.

Soloway and Ehrlich's model (also known as the domain model) emphasizes the highest level abstractions in the mental model. One aspect that sets this model apart from the others is the top-down development of the mental model with the assumption that the knowledge it uses has been previously acquired. By itself, this model does not take into account situations when code is novel and the programmer has no experience to use as a "backplane" in which to plug in new code.

Each of these models represent important aspects of code comprehension and many overlap in characteristics. For example, Brooks, Letovsky, and Shneiderman all focus on hierarchical layers in the mental representations. Brooks and Soloway & Ehrlich use a form of top-down program comprehension while Pennington use a bottom-up approach to code understanding. Letovsky and Shneiderman 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. However, we can take the best of these models, connect them and create an *Integrated Meta-model* that not only represents relevant portions of the individual models but also behaviors not found in them, e.g. when a programmer switches between top-down and bottom-up code comprehension.

Cognition models and the theories on which they are based must be grounded in experiments that either collected the information upon which a theory is based or validates it. The next section describes types of experiments commonly found in program cognition and important components of experimental design.

4 Experimental Paradigm

4.1 Components of Experimental Design

Program comprehension experiments have four major phases: *Definition*, *Planning*, *Operation*, and *Interpretation*. Several components further define each phase. Table 4 lists phases and components with a short explanation of each component.

Phase	*Components*	*Explanation*
Definition	Motivation/Purpose	Why is the experiment done.
	Object	What is being studied, e.g. corrective maintenance.
	Task	Specific task, e.g. debug module
Planning	Subjects	Expertise & quantity
	Language	Procedural/Declarative
	Code Size	Measured in Lines of Code
	Independent Variables	Manipulated factors
	Type	Hypothesis/Correlational/Naturalistic
	Measurement	What is being measured and how
Operation	Procedure	What was actually done during the experiment
Interpretation	Analysis	ANOVA, frequencies, etc.
	Validity	Where there any threats to validity?

Table 4: Components of Experimental Design

The *Definition* phase is the first part of any experiment. It defines why the experiment is important and what is being tested. We distinguish between *Motivation* and *Purpose*. An experiment's motivation is a high level description of the basis or justification for the study. The purpose of an experiment is a specific statement of goals and a *hypothesis*. A hypothesis is a precise statement of the question the experiment is to answer. The *Object* is the principal entity under study such as comprehension models and dynamic or static components of comprehension models (strategies and knowledge structures, respectively). The *Task* description specifies the precise activities of the subjects during the experiment. Task should be directly related to the object, for example if the object is to study the design process, the task may involve developing code during an experiment.

The *Planning* phase of an experiment involves *design* of the details of experimental procedure and how to *measure* the object under study. In program cognition, experimental design specifies *subjects*, (who is doing the task), *programming language and number of lines of code* (task descriptors), *independent variables* (measurable aspects of the task that relate to the hypothesis statement and can be measured with the procedures defined), and *type* of experiment.

All program comprehension experiments use and observe subjects. Subjects range from novice programmers to professional programmers. Important considerations include size of participant sample and level of expertise. The most commonly used languages in program comprehension experiments are Pascal and Fortran, with Cobol the next most frequently used. Differentiating between programming languages is important because comprehension may differ depending on the programming language used. Lines of code is another important descriptor of an experimental program comprehension task. Experiments try to determine qualitatively or quantitatively whether or to which degree specific factors (independent variables) affect an outcome (dependent variable).

Operation describes experimental procedure and events during experimentation. For example, if many of the subjects drop out of the experiment before it is completed, the results may be very different than if all subjects had stayed. This can affect interpretation.

Interpretation evaluates how well the experiment answered the original hypothesis. *Analysis* examines the measurement data of the dependent variables and draws conclusions on the results. *Validity* determines whether the results are sound. Validity is concerned with whether we measured the right thing in the right way. *Internal validity* refers to whether the independent variables actually *caused* changes in the dependent variables. *External validity* concerns the degree to which the conditions under which the data were collected are representative of those in the real world. It determines generality of the results.

As an example, Gellenbeck & Cook [15] conducted an experiment for which the *Motivation* was to understand what factors make one program easier to understand than another. The *Purpose* was to determine if typographic signalling makes programs easier to read and understand. The *Hypothesis* was that it does. The *Object* was the program understanding process while the *Task* was to understand a piece of code.

8 professional programmers were used as subjects. They tried to understand 913 LOC written in the C language. The *Independent Variables* consisted of the presence of a typographic signal (2 levels:Present/None), the type of module name (2 levels:Mnemonic/Neutral), and the presence of a header comment (2 levels:Present/None). The *type* of experiment was hypothesis testing. Measurements were taken on the accuracy of subject responses to questions (hypothesis questions) regarding the function of a particular routine (yes/no), subject confidence rating in accuracy to the hypothesis questions, time to respond to questions regarding the location of a routine that performed a specific function, and subject confidence in accuracy time to the location questions.

The *procedure* had the professional programmers study a 913-line hard copy of the program for 10 minutes. 8 versions contained the 8 possible combinations of typographic signaling, header comments, and mnemonic module names. This was followed by requiring the subjects to answer 24 hypothesis questions displayed on a computer terminal. After each question they had to rate, on a scale of 1...100, their own confidence in the answer they had just provided. Following the 24 hypothesis questions, the subjects answered 24 location questions. If the answer was not correct, the computer signalled an error and the subject was required to try again. The time to answer correctly was collected for each question.

The *analysis* showed that typographic signaling, header comments, and mnemonic names all helped in understanding code and that typographic signalling did not aid in locating information.

Figure 7 illustrates how phases are related to each other. The hypothesis is a refinement of the goals for the experiment. Hypotheses drive the remaining phases. Subjects, language, and code size are independent variables found in all program comprehension experiments. Independent and dependent variables are refinements of the hypothesis. Variables and the hypothesis prescribe the metrics and the measurement procedure necessary. For example, if the hypothesis is that experts recall critical lines of code faster than novices, then our independent variable is expertise, the metric is recall time in seconds or minutes, and the measurement procedure is to use a stopwatch and measure recall times for novices and experts. The actual operation of the experiment may not occur exactly as planned, so the analysis method can be affected by the procedure. For example, if subjects drop out of the experiment the analysis method must take into account missing data. The hypothesis is a precise statement of what is to be measured, and therefore also influences the analysis method. For example, hypotheses about whether two variables are related require correlational analyses of data collected during experimental operation.

Experimental design is an iterative process. If a hypothesis is not testable, the experiment needs redesigning starting with a restatement of the hypothesis. The next section describes in more detail each type of experiment and when it is appropriate.

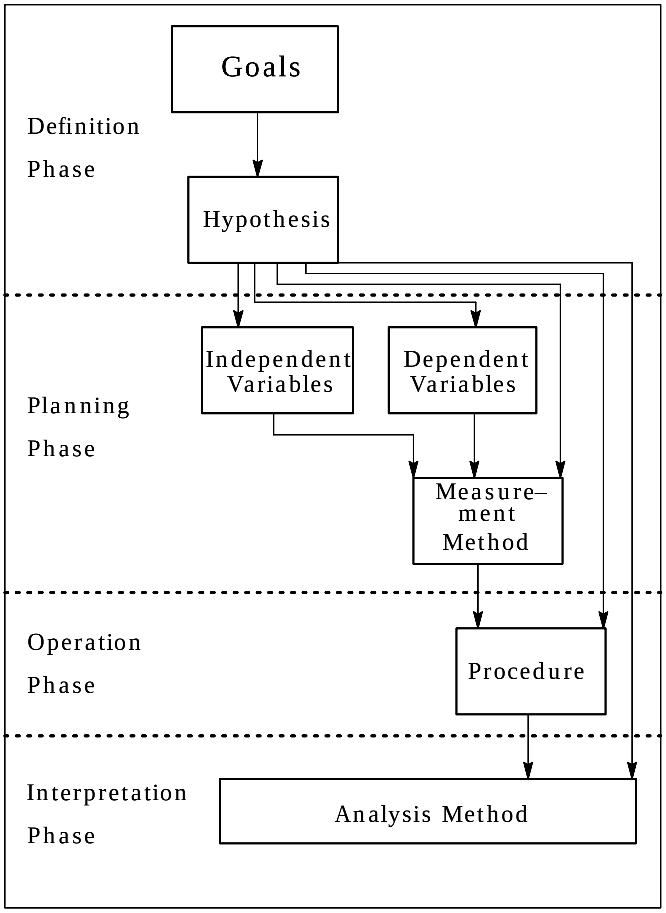

Figure 7: Interactions between Experimental Phases

4.2 Experimental Techniques

Objectives

Several types of experiments have been used for program comprehension experiments. The appropriate type depends on the purpose of the study and the amount of existing research. The 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. Here, more is known about the behavior in question, enough to explore relationships among variables. At the other end of the spectrum we find *hypothesis testing* experiments that investigate cause and effect between variables. Hypothesis testing are carefully controlled experiments whose purpose is to validate an existing theory. They can be designed once a theory exists and correlational studies indicate possible relationships between variables.

Field Studies – Naturalistic Observations, Interviews, Questionnaires

Field studies are exploratory data gathering techniques used in realistic settings. Naturalistic observations study behaviors as they occur, for example observing maintenance engineers as they try to understand code written by someone else. Observation is unobtrusive so that normal behaviors and events are not changed due to the presence of the observer. For example, Adelson & Soloway [2] conducted a field study of expert software designers in order to build a model of software design problem-solving skills. Three experts designed an Email system while observers watched and audio-taped think-aloud reports of the engineers during the design task. These reports were transcribed and analyzed. Analysis resulted in a model of problem solving skills for program design.

Interviews and questionnaires also collect information in a field setting. Both involve questions designed to elicit specific types of information. Interviews and questionnaires are *retrospective* because they report (after the fact) on information acquired some time before the interview or questionnaire.

Correlational Studies

Correlational observations try to determine if there is a relationship between variables. They are similar to hypothesis testing but do not explain cause and effect relationships. For example, Koenemann and Robertson [20] designed and executed an experiment to show that program comprehension is a hypothesis-driven, problem-solving process by determination of programmer preferred strategies. 12 professional programmers were assigned randomly to four different program modification tasks. Two types of strategies, opportunistic and systematic, were recognized for this experiment. The experimental hypothesis was that the relationship between the amount of time to finish a modification task and the number of pieces of information looked at determines the strategy used. Time to complete the modification task was correlated to the number of pieces of information programmers looked at. Results indicate a moderate relationship between modification time and number of pieces of information.

Hypothesis Testing

Hypothesis testing requires the formulation of a hypothesis which is then tested through observation and measurement. All dependent and independent variables are measurable and their levels controllable. Then, observed behavior can be explained in terms of the effect of dependent variable variations on levels of the independent variables. These tightly controlled experiments have high internal validity, but the need for such control may affect external validity. Since experiments are based on the statement of a hypothesis, this requires either an existing theory or prior observations that lead to a hypothesis. Most early program cognition experiments using hypothesis testing borrowed theoretical frameworks from psychology. *Quasi-experiments* (also known as *ex post facto studies*) are hypothesis testing experiments with the exception that subjects are not *randomly* assigned to levels of the independent variable. This is the case when preexisting differences define membership in different groups and the dependent variable is directly related to these groups.

For example, Adelson [1] designed a study in which she was interested in finding differences between expert and novice programmers in terms of behaviors and skills during code comprehension. Two groups of subjects consisted of novice and expert programmers. They studied the same code. The experiment measured the ability to recall lines of code verbatim and compared results between novices and experts. The experiment is an ex post facto study because there is no direct control over who belongs to each group, i.e. there is no random assignment of subjects.

By contrast, in the Gellenbeck & Cook study, all subjects were professional programmers with approximately the same amount of experience. Subjects were randomly assigned to one of eight different versions of code. Each routine within a version contained a randomly assigned condition (i.e. Mnemonic Name, no header comment, typographic signaling). This design compensates for effects caused by the individual differences between subjects by random assignment of subject to version and is therefore a hypothesis testing experiment.

4.3 Program Understanding Experiments

Tables 5, 6, and 7 present a list of program understanding experiments by experimental technique. Depending on the type of experiment, the results of each support some portion of the associated comprehension model. While this is not an exhaustive list, it is intended to show the variety of experimental procedures, tasks, participants, and programming languages that have been used to develop knowledge of program understanding for maintenance and general code understanding.

Maintenance Task columns describe the overall task focus for each experiment. In all cases, program understanding behavior is being measured in the context of a larger task, for example corrective maintenance. *Understand*ing is a maintenance task in and of itself when responsibility for maintaining code is first assigned. The person assuming the responsibility will want to understand (at least at a high level) the code before performing any other maintenance on it.

The *LOC* columns describe the size of the code used during the task in terms of number of lines of code. The *Subjects/Expertise* columns report the number and type of subjects. Novice participants are typically first year programming students. Intermediate subjects are also usually students who have had slightly more experience than novice, i.e. second or third year computer science students. Experts come from either a professional or computer science graduate student population. *Experimental Design or Analysis* briefly describes the object and analysis method of the study. *Purpose* states the purpose of the experiment.

Maintenance Task	*LOC*	*Subjects/ Expertise*	*Language*	*Experiment Design or Analysis*	*Purpose*	*Cite*
Understand	17	22 Grad	Pascal	Protocol Analysis [Strategies]	Collect data on strategies used by experts	[12]
	67	42 Novice 58 Experts	Cobol	Protocol Analysis [Knowledge Structures]	Investigate nature of expert chunks	[35]
Enhance	250	6 Expert	Fortran	Protocol Analysis [Hypotheses]	Study question asking & hypotheses	[21]
	250	10 Expert	Fortran	Protocol Analysis [Strategies]	Show success/failure of task related to strategies	[22]
	1057	12 Grad	C	Protocol Analysis [Behaviors]	Book-paradigm using beacons better than source listings	[23]
	200	40 Expert	Cobol & Fortran	Protocol Analysis [Strategies]	Study role of knowledge in programming	[25]
	250	20+ Expert	Fortran	Protocol Analysis [Strategies]	Determine types of representations help programmers	[33]
Code	N/A	1 Expert	Fortran	Protocol Analysis [Design Behavior]	Build model of expert information processing	[2]
	15	83 Novice	Pascal	Protocol Analysis [Plans]	Do novices have plans	[29]
Design	N/A	2 Expert	Pseudocode	Protocol Analysis [Design Behavior]	Build model of expert problem solving skills	[2]
	22	49 Expert 90 Interm 64 Novice	Pascal	Protocol Analysis [Looping Strategies]	Impact of looping strategies on use of looping constructs	[30]
Debug	350	8 Expert 8 Novice	Cobol	Protocol Analysis [Debugging Behavior]	Determine differences in debugging between novice & experts	[34]
Understand, Corrective, Adaptation, & Perfective	50-80,000+	11 Expert	C, C-shell, Make	Protocol Analysis [Strategies], [Processes],[Actions] [Episodes]	Build a theory of program comprehension on large-scale code	[36]

Table 5: Observational Studies – Comprehension Experiments

Table 5 lists observational studies. *Protocol analysis* refers to an audio- and/or video-taped session that is transcribed and analyzed. Protocols are typically think-aloud reports of subjects working on a task. Languages are Pascal, Cobol, and Fortran. We find a large range of number of subjects. As the number of subjects increases, the LOC decreases. This is understandable because the amount of time for 90 subjects to understand a large piece of code makes such experiments prohibitively large.

Maintenance Task	*LOC*	*Subjects/ Expertise*	*Language*	*Experiment Design or Analysis*	*Purpose*	*Cite*
Understand	16	5 Novice 5 Expert	PPL	Expertise vs. chunks recalled	Are experts' chunks different from novices'	[1]
	42	10 Novice 7 Grad	Pascal	Cluster Analysis	Are experts' plans different from novices'	[26]
Adapt & Enhance	600	12 Expert	Pascal	Pieces of info examined vs. modify time	What strategies do experts use.	[20]

Table 6: Correlational Studies – Comprehension Experiments

Table 6 lists correlational experiments. The list is small. The experiments concentrate on differences between novice and expert programmer behavior. LOC and number of subjects is small. Two used Pascal. One used PPL (Polymorphic Programming Language), a language developed to have properties in common with APL and PL/I.

Table 7 lists hypothesis testing and quasi- experiments. Experimental design or analysis consists of several different techniques. *Recall* experiments have subjects study code for a given period of time and then ask them to recall it from memory. *Verbatim* recall recalls code *exactly* as it appeared while *Free* recall requires subjects to describe what appeared in less specific terms. *Cloze tests* have subjects study code containing a missing line or word and require them to "fill in the blank". Most experiments measure time. For example, *eye fixation time* is the time spent looking at one code entity; *Time between plan jumps* is time spent within a single plan and between different plans; and *Response time to questions* tracks the time it takes to press a "Y" or "N" key on the keyboard in response to a question.

A majority of these experiments have been done using general understanding tasks and languages such as Pascal, Cobol, Fortran, or Basic. Most of these experiments use very small-scale code. Comparing the typical number of subjects in these experiments to those in the observational or correlational categories shows that hypothesis testing experiments use many more subjects. The combination of these factors demonstrate several aspects of tightly controlled experiments. The function of a hypothesis testing experiment is to verify a detailed hypothesis, the basis of which is grounded in previous research. Observational and correlational studies may have been done to construct the hypothesis. Therefore, more subjects are needed for verification if the results are going to generalize to a larger population. Concrete measurements such as time and counts are also necessary since better statistical methods exist for finding cause and effect on these types of data.

5 Issues

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 8 reorganizes the experiments from tables 5, 6, and 7 in terms of models, common elements of cognition, and code size, programming language, and subjects. For code size, *small* applies to programs of less than 900 lines of code (LOC). Medium size code refers to code between 900 and 40,000 LOC. Large scale code contains more than 40,000 LOC. The language columns distinguish between languages such as Cobol, Fortran, Basic, and Pascal and state of the art development environments such as C/Unix with tools like *Make* or *lint*. Subjects are categorized as novice, grad students, or professional programmers. Each cell in the table represents experiments that investigated the row component with the column attribute.

Maintenance Task	*LOC*	*Subjects/ Expertise*	*Language*	*Experiment Design or Analysis*	*Purpose*	*Cite*
Under-stand	10	10 Novice 9 Grads	Pascal	Eye Fixation Time	Does experience affect code examination time	[7]
	66	23 Novice	Pascal	Reaction time to questions re:code	Difference between graphical & textual novice representations	[8]
	25	96 Grads	Pascal	Correct ID of procedure function	Do meaningful identifier names act as beacons.	[14]
	913	8 Expert	C	Response time to questions	Does typographic signalling make programs easier to understand.	[15]
	15	80 Expert	Cobol & Fortran	Response time to questions	Do programmers have text structure & plan knowledge.	[25]
	15	94 Novice 45 Interm	Pascal	Cloze Test Fill in line	Do experts have plans & rules of discourse.	[31]
	23	12 Novice 12 Grads	Pascal	Verbatim Recall	Are code lines that swap values beacons for sorting.	[39]
Debug	101	48 Interm	Fortran	Time to locate bug	Assess debugging framework	[3]
	11	72 Novice	Basic	Error Count	Are plans related to design method	[11]
	30	48 Novice	Basic	Recall on critical lines	Are plans related to design method	[11]
	350	8 Novice 8 Expert	Cobol	Time to find error, counts	Expert & novice differences in debugging.	[34]
Modify	373	18 Novice 18 Expert	Pascal	Free Recall	Does program structure affect mental model	[4]
Design Code	40	12 Novice 12 Interm 12 Expert	Pascal & Basic	Time between Plan jumps	Effects of language on strategy Development	[10]
Enhance	1000	53 Grads	Pascal	Time to finish task	Book-Paradigm using beacons helps understanding	[23]

Table 7: Hypothesis Testing & Quasi-Experimental Comprehension Experiments

Table 8 clearly points to the "white areas" on the map 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, appears in only three experiments. When we look at the object of the study, most investigate strategies, beacons, or plans. Few studies exist of processes, rules of discourse, actions, program comprehension episodes, and of entire cognition models. In light of this information, program cognition must concentrate on three issues:

1. *Scalability of experiments*
   We must investigate whether the many well designed experiments using small-scale code scale up for production code. For example, Vessey's study of expert programmer's knowledge [35] and Pennington's study of mental representations of code [24] used programs of lengths varying between 67 LOC and 200 LOC. These studies are appropriate for answering questions about understanding small program segments or very small programs. However, we cannot say anything about the interactions of these isolated components of understanding nor whether these results will play an important role in understanding large programs.

2. *Static versus Dynamic Behavior*
   Current results mainly focus on the *static properties* of programming skills [9]. For example, experiments identified persistent knowledge such as searching or sorting routines, but they do not investigate

knowledge use and application.

3. *Theory Building*
   Many experiments are designed to measure specific conditions (e.g. do programmers use plans?) but the experimental hypotheses (programmers use plans when understanding code they have never seen before) are not based on a well-defined theory of program comprehension. Sheil [27] concludes that "Our primary need at the moment is for a theory of programming skill that can provide both general guidance for system designers and specific guidance to psychologists selecting topics for detailed studies. The experimental investigation of such factors as the style of conditional notation is premature without some theory which gives some account of why they might be significant factors in programmer behavior." Although this paper was written in 1981, in the past 12 years very few theories concerning program comprehension have been advanced. Theories regarding large scale program comprehension for specialized maintenance tasks are in their infancy.

6 Conclusion

Program understanding is a key factor in software maintenance and evolution. This paper summarized major elements of cognition, how they are represented in several different models of program cognition, and the importance of experimentation in developing model theories and in validating them. Several conclusions can be drawn from this survey:

1. While a great deal of important work exists, most of it centers around general understanding and small-scale code.

2. Existing results from related areas need further investigation. For example, Pennington's model borrows from work in understanding stories. Perception and problem solving are two areas that are strongly related to program comprehension.

3. Some of the results generalize or appear as components of larger results. For example, elements of Pennington's and Soloway & Erhlich's models appear in the integrated Meta-Model.

4. Availability of data is a challenge. Obtaining expert software engineers working on production code is difficult unless the companies that maintain large-scale code encourage their maintenance engineers to participate in program comprehension experiments. Yet these work situations are inadequately addressed by current experiments.

5. The literature fails to provide a clear picture of comprehension processes based on specialized maintenance tasks like adaptive or perfective maintenance. While models of the general understanding process play a very important part in furthering insight into complete understanding of a piece of code, they may not always be representative for narrow tasks like reuse or enhancements which may more efficiently employ strategies geared towards partial understanding.

We still have much to learn about how programmers understand code and how much understanding is necessary. A better grasp of how programmers understand code and what is most efficient and effective can lead to a variety of improvements: better tools, better maintenance guidelines and processes, and documentation that supports the cognitive process.

References

[1] Beth Adelson, **Problem solving and the development of abstract categories in programming languages**, In: Memory and Cognition, 1981,Vol. 9(4), pp. 422-433.

[2] Beth Adelson and Elliot Soloway, **A Model of Software Design**, In: The Nature of Expertise, M.Chi, R. Glaser, and M.Farr (Eds), ©1988, Lawrence Erlbaum Associates, Publishers, pp. 185-208.

[3] Michael E. Atwood and H. Rudy Ramsey, **Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging**, In: Technical Report # TR-78-A21, August 1978, Science Applications, Inc.

[4] Deborah A. Boehm-Davis, Robert W. Holt, and Alan C. Schultz, **The role of program structure in software maintenance**, In: International Journal of Man-Machine Studies, 36(1992), pp. 21-63.

[5] Ruven Brooks,, **Towards a theory of the cognitive processes in computer programming**, In: International Journal of Man-Machine Studies, 9(1977), pp. 737-751.

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

[7] Martha E. Crosby and Jan Stelovsky, **How Do We Read Algorithms? A Case Study**, In: IEEE Computer, January 1990, pp. 24 - 35.

[8] Nancy Cunniff and Robert P. Taylor, **Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension**, In: Empirical Studies of Programmers:Second Workshop, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 114 - 131.

[9] Simon P. Davies, **Models and theories of programming strategy**, In: International Journal of Man-Machine Studies, 39(1993), pp. 237 - 267.

[10] Simon P. Davies, **The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior**, In: Cognitive Science, Vol.15 No. 4, October - December, 1991, pp. 547 - 572.

[11] Simon P. Davies, **The nature and development of programming plans**, In: International Journal of Man-Machine Studies, 32(1990), pp. 461 - 481.

[12] Francoise Detienne and Elliot Soloway, **An empirically-derived control structure for the process of program understanding**, In: International Journal of Man-Machine Studies, 33(1990), pp. 323-342.

[13] Francoise Detienne, **Program Understanding and Knowledge Organization: The Influence of Acquired Schemata**, In: Cognitive Ergonomics: Understanding, Learning, and Designing Human-Computer Interaction, ©1990, Academic Press, pp. 245-256.

[14] Edward M. Gellenbeck and Curtis R. Cook, **An Investigation of Procedure and Variable Names as Beacons during Program Comprehension**, Tech Report 91-60-2, Oregon State University, 1991.

[15] Edward M. Gellenbeck and Curtis R. Cook, **Does Signaling Help Professional Programmers Read and Understand Computer Programs?**, Tech Report 91-60-3, Oregon State University, 1991.

[16] 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, ©1987, Ablex Publishing Corporation, pp. 65 - 82.

[17] Raymonde Guindon, **Knowledge exploited by experts during software systems design**, In: International Journal of Man-Machine Studies, 33(1990), pp. 279-182.

[18] K. C. Kang, **A Reuse-Based Software Development Methodology**, In: Proceedings of the Workshop on Software Reuse, G. Booch and L. Williams, eds., Rocky Mountain Inst. of Software Engineering, SEI, MCC, Software Productivity Consortium, Boulder Colorado, October 1987.

[19] Walter Kintsch and Teun A. van Dijk, **Toward a Model of Text Comprehension and Production**, In: Psychological Review, 85(5), 1978, pp. 363 - 394.

[20] Jurgen Koenemann and Scott P. Robertson, **Expert Problem Solving Strategies for Program Comprehension**, In: ACM? March 1991, pp. 125-130.

[21] Stanley Letovsky, **Cognitive Processes in Program Comprehension**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 58 - 79.

[22] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway, **Mental Models and Software Maintenance**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 80 - 98.

[23] Paul W. Oman and Curtis R. Cook, **The Book Paradigm for Improved Maintenance**, In: IEEE Software, January 1990, pp. 39-45.

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

[25] Nancy Pennington, **Comprehension Strategies in Programming**, In: Empirical Studies of Programmers:Second Workshop, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 100 - 112.

[26] Robert S. Rist, **Plans in Programming: Definition, Demonstration, and Development**, In: Empirical Studies of Programmers: 1st Workshop, 1986, Washington, D.C., pp. 28-47.

[27] B.A. Sheil, **The Psychological Study of programming**, In: ACM Computing Surveys, March 1981, Vol13, pp. 101 - 120.

[28] Ben Shneiderman and Richard Mayer, **Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results**, In: International Journal of Computer and Information Sciences, 1979, Vol.8, No.3, pg. 219-238.

[29] Elliot Soloway, Kate Ehrlich, Jeffrey Bonar, and Judith Greenspan, **What Do Novices Know About Programming?**, In: Directions in Human/Computer Interaction, Albert Badre and Ben Shneiderman (Eds), ©1982, ALBEX Publishing Corp., pp. 27-54.

[30] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich, **Cognitive Strategies and Looping Constructs: An Empirical Study**, In: Communications of the ACM, November 1983, 26(11), pp. 853-860.

[31] Elliot Soloway and Kate Ehrlich, **Empirical Studies of Programming Knowledge**, In: IEEE Transactions on Software Engineering, September 1984, Vol. SE-10, No. 5, pp. 595-609.

[32] 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, ©1988, ALawrence Erlbaum Associates, Publishers, pp. 129-152.

[33] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert, **Designing Documentation To Compensate For Delocalized Plans**, In: Communications of The ACM, Vol. 31, No. 11, November 1988, pp. 1259-1267.

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

[35] Iris Vessey, **On matching programmers' chunks with program structures: An empirical investigation**, In: International Journal of Man-Machine Studies, (1987)27, pp.65-89.

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

[37] 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, July 1993, pp. 230 - 239.

[38] 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, July 1993, pp. 78 -86.

[39] Susan Wiedenbeck, **Processes in Computer Program Comprehension**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 48 - 57.

Component	*Code Size: Small*	*Code Size: Medium*	*Code Size: Large*	*Lang.=Pascal, Fortran,Cobol*	*Lang.=C, Environ.*	*Subject = Novice*	*Subject = Grad Student*	*Subject = Professional*
Top-Down			[37]		[37]			[2],[37]
Situation	[24]		[37]	[24]	[37]			[24],[37]
Program	[24]		[37]	[24]	[37]			[24],[37]
Other	[34],[21]			[34],[5],[21]		[34]		[34],[5],[21]
Processes			[36]		[36]		[36]	
Hypotheses	[21],[34]			[21],[34]		[34]		[21],[34]
Strategies	[12],[20], [22], [25], [30],[33]			[12],[20],[22], [25],[30],[33]		[30]	[12],[30]	[20],[22],[25], [30], [33]
Plans	[10],[11], [26],[29], [31]			[10],[11],[26], [29],[31]		[10],[26], [29],[31]	[10],[11],[26], [31]	[10],[13]
Rules of Discourse	[31]			[31]		[31]	[31]	
Chunks	[1],[4], [35]			[1],[4],[35]		[1]		[1],[4],[35]
Episodes	[34]		[36]	[34]	[36]	[34]		[34],[36]
Beacons	[14], [39]	[15],[23]		[14],[23],[39]	[15],[23]	[39]	[14],[23],[39]	[15]
Text Structures	[24]			[24]				[24]
Actions	[21]		[36]	[21]	[36]			[21],[36]

Table 8: Component × Experimental Attribute Table