8

PROGRAM COMPREHENSION

The most incomprehensible thing about the world is that it is comprehensible.
— Albert Einstein

8.1 GENERAL IDEA

It is important to comprehend the details of any complex artifact to be able to maintain it, because maintenance requires modifications to portions of the systems—and maintaining a software system is no different. Inaccurate and incomplete understanding of a software system before performing a modification on it is likely to severely degrade its performance and reliability. Therefore, good program comprehension plays an important role in providing effective software maintenance and enabling successful evolution of software systems. In order to get a handle on the role of program comprehension, we look at the five types of tasks commonly associated with software maintenance and evolution and the types of activities involving those tasks. As listed in Table 8.1, the five kinds of maintenance and evolution tasks are *adaptive*, *perfective*, and *corrective* maintenance; *reuse*; and *code leverage*. The detailed activities associated with each of the five tasks have been listed in the second column of Table 8.1. Table 8.1 indicates that understanding the system or problem is common to all maintenance and evolution tasks. It may be noted that understanding of a system is a cognitive issue, and to study the cognitive processes behind the tasks in Table 8.1, a number of cognition models have been developed, as listed in Table 8.2.

Software Evolution and Maintenance: A Practitioner's Approach, First Edition.
Priyadarshi Tripathy and Kshirasagar Naik.

TABLE 8.1 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

Source: From Reference 1. © 1995 IEEE.

TABLE 8.2 Code Cognition Models

Model	Maintenance Activity	Authors
Control flow	Understand	Pennington
Functional	Understand	Pennington
Top-down	Understand	Soloway, Adelson, and Ehrlich
Integrated	Understand, Corrective, Adaptive, and Perfective	Von Mayrhauser and Vans
Other	Enhancement	Letovsky
	Understand	Brooks Shneiderman and Mayer

Source: From Reference 1. © 1995 IEEE.

8.2 BASIC TERMS

Some researchers use the term *maintainer* to refer to the person who is maintaining a software system. They want to differentiate a maintainer from a programmer. In order to understand the cognition models of Table 8.2, we discuss the following set of terms in the following subsections:

- Goal of code cognition
- Knowledge
- Mental model

8.2.1 Goal of Code Cognition

A maintainer sets out to understand a program with a specific goal in her mind. An example of specific goal is debugging the program to detect a fault, that is, the cause of a known failure. Another example is adding a new function to the existing system. Identifying the goals can help in defining the scope of program comprehension, where scope refers to complete or partial understanding of code.

A program comprehension process is a sequence of activities that use existing knowledge about a system to acquire new knowledge that ultimately meets the goals of a code comprehension task. In other words, program comprehension is a process of knowledge acquisition about the program. The different kinds of knowledge associated with program comprehension are explained below.

8.2.2 Knowledge

As software specialists, programmers possess two kinds of knowledge:

- General knowledge: This covers a broad range of topics in computer systems and software development. A non-exhaustive list of topics is: algorithms and data structures, operating systems, programming principles, programming languages, programming environment, software architectures and design techniques, testing and debugging techniques, different solution approaches, and computer networks. More general knowledge enable a programmer to gain more software-specific knowledge.
- Software-specific knowledge: This represents an understanding of the details of the software to be comprehended. The details of a software system include the application domain, the problem being solved by the software, and the solution strategy adopted in the software system.

While gaining software-specific knowledge during the process of program comprehension, programmers may find the need to have more general knowledge about computer systems and, therefore, acquire the relevant general knowledge by referring to other sources. For example, in order to comprehend the performance aspect of a wireless Internet-based application, a programmer may want to know some details of the medium access control (MAC) protocols for wireless local area networks

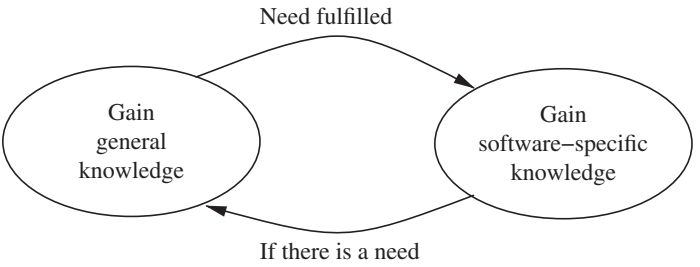

FIGURE 8.1 Gaining general knowledge and software-specific knowledge

(WLAN). The general knowledge may involve the concept of maximum data packet size, for example. The programmer may also want to learn how the transmission control protocol's (TCP) congestion control mechanism affects data delivery performance. Therefore, there is a need for a programmer to go back and forth between acquiring general knowledge and software-specific knowledge, as illustrated in Figure 8.1. Some examples of software-specific knowledge are as follows:

- The software system implements public-key cryptography for data encryption.
- The software system has been structured as a three-tier client–server architecture.
- Module *x* implements a location server.
- A certain *for* loop in method *y* may execute for a random number of times.
- Variable *mcount* keeps track of the number of times module *z* is invoked.

In the process of gaining new knowledge about a software system, a programmer essentially learns the details concerning the following aspects:

- Functionality: This includes an understanding of the functional and non-functional requirements of the software system.
- Software architecture: Knowledge about a system's software architecture leads to further inquisitions about the system. For example, if a programmer knows that the system has been designed with a three-tier client/server architecture, then he will try to know more about distribution of functionalities, assignment of resources to services, and so on.
- Control flow and data flow: Understanding the control flow in a system helps comprehend the execution sequences of the modules. The programmer gets to know the sequences in which data are accessed and updated.
- Exception handling: The programmer needs to know what actions the software system takes in response to exceptions, such as timeout, unavailability of run-time resources, and execution errors, to name a few.
- Stable storage: This refers to the means for storing intermediate results in non-volatile memory so that program execution can be resumed at a point where failure occurred.

- Implementation details: The programmer may want to know what algorithms have been chosen and how those have been implemented. Moreover, the programmer may want to know the functionality and data scope of objects and their implementation details.

8.2.3 Mental Model

The concept of *mental model* [2] is frequently used in the discussion of program comprehension. A mental model describes a programmer's mental representation of the program to be comprehended. Consequently, a mental model of a program is not unique. Rather, different programmers may have different mental models depending upon their own knowledge and interpretation about the same program. A programmer develops a mental model of a software by identifying both *static* and *dynamic* elements of the software system. For example, a *for* loop is an element of a program. A system call to open a TCP connection is a program element. Similarly, the overall control flow is a program element. A programmer must have a good understanding of the individual program elements as well as how some of them interact.

A. Static Elements of the Mental Model The static elements of a program are as follows:

- Text-structures
- Chunks
- Schemas
- Plans
- Hypotheses

Text-structure: Code text and its structure are known as text-structure. Text-structure knowledge is useful in gaining control flow knowledge for program understanding. A programmer can easily identify the following kinds of text-structures: (i) *loop* constructs, namely, *for*, *while*, and *until*; (ii) *sequences*; (iii) *conditional* constructs, namely, *if-then-else*; (iv) *variable* definitions and initializations; (v) *calling* hierarchies within and among modules; and (vi) definitions of module parameters. A programmer can examine text-structure at the statement level and for their dependencies. Understanding text-structure is the beginning of program comprehension.

Chunk: The concept of program chunks enables programmers to create higher level abstractions from lower level abstractions. A program chunk is a block of related code segments. For example, a code block initializing a module's parameters is a useful chunk that tells the programmer about the nature of the parameters and their value ranges. A *for* loop is another example of code chunk, whose understanding will create an abstraction of the loop as an internal functional step performed by the program.

Schema: Research on text comprehension has produced an important concept called *schema*. Schemas are *generic knowledge structures* that guide the comprehender's interpretations, inferences, expectations, and attention when passages are comprehended [3]. The concept of programming plans, to be explained in this chapter, corresponds to the above notion of schemas [4].

Plans: Plans are broad kinds of *knowledge elements* used by programmers in order to comprehend programs. A knowledge element is anything that is useful in understanding a program. For example, if the name of a function gives an indication of the activity performed by the function, then the function identifier is a knowledge element. A second example of knowledge element is a block of comments describing a `for()` loop. In addition, a `for()` loop itself is a knowledge element, because a programmer can read the chunk of code representing the loop and get a good idea about what the code does. A third example of knowledge element is the description of the problem domain of the program. Plans easily attract the attention of programmers while they are reading a program and its documentations. A doublylinked list and a heap data structure are examples of plans. The designer has planned to implement certain concepts with those data structures. A plan is a kind of schema with two parts: *slot type* and *slot filler*. Slot types describe generic objects, whereas slot fillers are customized to hold elements of particular types. For example, a tree data structure is a generic slot type, whereas a code segment, such as a `for()` loop's code, is an example of a slot filler. The programmer links the slot-type and slot-filler structures by means of the widely used modeling relationships, namely, *kind-of* and *is-a*.

There are two broad kinds of plans as follows:

- *Domain plans:* These include knowledge about the real-world problem, including its environment, that the software system is solving. For example, if a software system is for numerical analysis application, plans to develop a software package will include schemas for different aspects of linear algebra, such as matrix multiplication and matrix inversion. As another example, if a software system is for voice-over-IP (Internet Protocol) applications, plans to develop the package will include schemas for locating destinations and setting up a call. Domain plans help programmers understand the code. For example, without a basic understanding of how destination users are located before setting up a call, code comprehension will not be accurate.
- *Programming plans:* Programming plans are *program fragments* representing action sequences that programmers repeatedly apply while coding. A repeated action sequence is also known as a stereotypic action sequence. A programmer may design a `for()` loop to search an item in a data set and repeatedly use the loop in many places in the program. Such a `for()` loop is an example of a programming plan to implement the system. Programming plans differ in their granularities to support low level or high level tasks.

Hypotheses: As programmers read code and the related documents, they start developing an understanding of the program to varying degrees. The qualities of their understanding can be characterized in terms of depth, breadth, and degree of accuracy.

Programmers can test the results of their understanding as *conjectures*, which are also called *hypotheses* by Letovsky [5]. The three kinds of conjectures are as follows:

- **Why:** *Why* conjectures hypothesize the *purpose* of a program element. A program element can be a function, a procedure, a design choice, or even a conditional statement at the lowest level of code. Verification of a *why* conjecture allows the programmer to have a good understanding of the purpose of the program element.
- **How:** *How* conjectures hypothesize the *method* for realizing a program goal. Given a program goal, say, as a deliverable functionality, the programmer needs to understand how that goal has been accomplished in the program.
- **What:** *What* conjectures enable programmers to classify program elements. For example, when a programmer finds an identifier, she should be able to classify it according to known, high level types, such as, constants, variables, and functions. Further classification of identifiers can be done in terms of their functions. For example, a variable called *TCPcon1* may hold a TCP connection identifier, and another variable called *RetransTimer* may hold a timer for packet retransmission.

A conjecture may not be completely correct because of the incomplete understanding of code by the programmer. Therefore, it is useful to talk about the degree of certainty of conjectures. At the beginning of the comprehension process, a programmer might make uncertain guesses, whereas she may be able to make almost certain conclusions after having a good comprehension of the program. Hypotheses are a way for a programmer to understand code in an incremental manner. After some understanding of the code, the programmer forms a hypothesis and verifies it by reading code. Verification of a hypothesis results in either accepting the hypothesis or rejecting it. By continuously formulating new hypotheses and verifying them, the programmer understands more and more code and in increasing details.

B. Dynamic Elements of the Mental Model The dynamic elements of a program are as follows:

- Chunking
- Cross-referencing
- Strategies

Chunking: Recall that lowest level chunks are code fragments representing low level functionalities. In order to understand a program in terms of its higher level functionalities, a programmer creates higher level abstraction structures by combining lower level chunks. This process of creating higher level chunks is called chunking. Note that chunking is not a one-step process. Rather, the concept of chunking can be repeatedly applied to create increasingly higher levels of abstractions. When a block of code is recognized, it is replaced by the programmer with a *label* representing

the functionality of the code block. Similarly, a block of lower level labels can be replaced with one higher level label representing a higher level functionality.

Cross-referencing: Cross-referencing means being able to link elements of different abstraction levels. This helps in building a mental model of the program under study. For example, control flow and data flow can be program elements at a lower level, whereas functionalities are higher level program elements—and there is a need for being able to make links between control flow and data flow elements and program functionalities.

Strategies: Intuitively, a strategy is a planned sequence of actions to reach a specific goal. A strategy is formulated by identifying actions to achieve a goal. For example, if the goal is to understand the code representing a function, one can define a strategy as follows. Understand the overall computational functionality of the function, that is, what the function is expected to compute, by reading its specification, if it exists; understand all the input parameters to the function; read all code line-by-line; identify chunks of related code; and create a higher level model of the function in terms of the chunks. Creating a higher level model from lower level models requires cross-referencing, as explained before. In other words, strategies guide the two dynamic elements, namely, chunking and cross-referencing, to produce higher level abstraction structures.

8.2.4 Understanding Code

The two key factors influencing code understanding are: (i) acquiring knowledge from code; and (ii) the level of expertise of the code reader. Code is a rich source of information to understand it, while the level of expertise determines how quickly the code is understood. Experienced programmers can manipulate the information available in code more easily and quickly than novice programmers to construct more accurate mental models. These two factors are explained in detail in the following.

A. Acquiring Knowledge from Code Several concepts can be applied while reading code in order to gain a high level understanding of programs. In the absence of those concepts, code comprehension can be a difficult task. In what follows, we explain those concepts.

- *Beacons:* A beacon is code text that gives us a cue to the computation being performed in a code block. For example, meaningful procedure names, such as `swap()`, `sort()`, `select()`, `startTimer()`, and `openTCPconnection` give us a high level understanding of code blocks. Code with good quality beacons are easier to understand. By good quality beacons we mean whether or not the beacons precisely reflect the code functionalities. For example the beacon *sendTCPpackets* is more useful than the beacon *sendpackets*, because the former indicates that data packets are sent over a TCP connection.
- *Rules of programming discourse:* Rules of programming discourse specify the conventions, also called "rules," that programmers follow while writing code.

The rules are increasingly followed as programmers become more experienced. Some examples of those rules are the following:

– *Function name:* The function name agrees with what the function does. For example, function names `sort()`, `delete()`, `startTimer()`, and `openTCPconnection()` are clearly understood, and those functions should perform tasks implied by their names. Moreover, a function should return the kind of arguments implied by its name. For example, `sort()` should return a sorted data structure, `delete()` should return OK status, `startTimer()` should return a timer object, and `openTCPconnection()` should return an opened TCP connection identifier.
– *Variable name:* Choose meaningful names for variables and constants. By looking at the name of a variable, one should be reasonably clear about the purpose of the variable. For example, the variable `count` indicates that the programmer is counting something. Similarly, the integer constant `MAX_USERS` gives an indication that the maximum number of users supported by the system is given by the constant `MAX_USERS`.

Essentially, the rules set up expectations in the minds of a reader about what should be in the program. In addition, it is assumed that programs are composed from programming plans governed by rules of programming discourse. Otherwise, a correct program from the viewpoint of solving the problem can be difficult for a programmer to write and difficult for a reader to comprehend.

B. Levels of Expertise of Code Readers Programmer experience and domain knowledge have a positive impact on code comprehension. Expert programmers tend to possess the following characteristics [6, 7]:

- *Organization of knowledge by functional characteristics:* A novice programmer is likely to organize program knowledge in terms of the program syntax, whereas experts organize knowledge in terms of algorithms and program functionalities. Consequently, the constructs used by experts are succinct and more informative than the program syntax used by novice programmers. In other words, expert programmers create a complementing view of the program, rather than what is plainly seen as a result of using program syntax.
- *Comprehension with flexibility:* Experts tend to generate a breadth-first view of the program, and keep adding useful details as more information is available during the understanding process. This is due to their better understanding of the program and their ability to more quickly discard invalid hypotheses and assumptions than novice programmers.
- *Development of specialized design schemas:* A schema is a higher order knowledge structure for describing program behavior in a specific domain of activity [8]. The knowledge structure specifies principal elements in a domain and includes mechanisms which facilitate the comprehension and generation processes. It is used to organize complex entities or concepts into constituents. An example of a specialized design schema is the *divide-and-conquer* schema

[7]. Using the divide-and-conquer schema, a programmer can divide a system into subsystems, and a problem into subproblems. Design schemas can vary in complexity and granularity, and, even some schemas can be simple rules: if the caller and the callee are independent processes, let them communicate by passing messages.

8.3 COGNITION MODELS FOR PROGRAM UNDERSTANDING

Research literature on software engineering is replete with cognition models. In this section, we explain the commonly referenced models one-by-one:

- Letovsky model
- Shneiderman and Mayer model
- Brooks model
- Soloway, Adelson, and Ehrlich model (top-down model)
- Pennington model (bottom-up model)
- Integrated metamodel

8.3.1 Letovsky Model

Letovsky's program comprehension model has been shown in Figure 8.2 [5]. The model consists of five main components: a knowledge base, a mental model, an assimilation process, some external representations about the software system, and, possibly, a dangling purpose unit. In the following we explain the above five components one by one.

1. **Knowledge base:** A programmer needs to assimilate a large body of knowledge to create a mental model of the software system under consideration. A

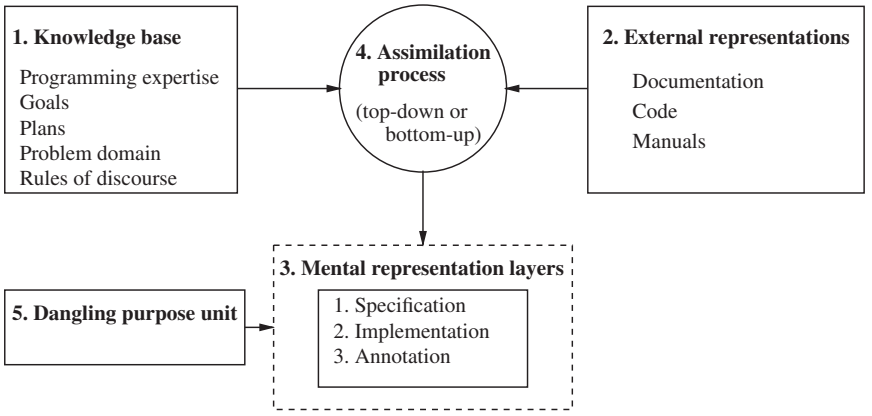

FIGURE 8.2 Letovsky's program comprehension model

knowledge base is a body of information about the program, and it includes the following:

- *Programming expertise:* Programming expertise is useful in asking probing *questions*, making *conjectures* to create abstractions from low level details by connecting observed code details with known program goals, and *searching* for specific information in the code. Conjectures are also called *hypotheses* [1]. The questions are grouped into five categories: *why*, *how*, *what*, *whether*, and *discrepancy*. *Why* questions are designed to know about the purpose of actions and design choices. *How* questions assist the programmer to learn about the way some goal of the code is accomplished. *What* questions are used to find out what a variable or code fragment is. *Whether* questions are asked to know if the code behaves in a certain way. *Discrepancy* questions are meant for resolving confusions and apparent inconsistencies in the code. Some example questions are: *Why is the variable being reset to zero?*, *What is being done to the memory block after the data is transmitted?*, and *Why (confusion) is the memory block being deleted in two places?* After asking questions, programmers often conjecture answers to their own questions based on their current understanding of the program. Similar to the classification of questions, programmers design *why*, *how*, and *what* conjectures. A *why* conjecture is made to assert an understanding of the purpose of an action or a design choice. A *how* conjecture is constructed to show an understanding of the way some goal of the program is accomplished. A *what* conjecture is formulated to demonstrate an understanding of what a variable or code segment does. Some conjectures could be pure guesses, because insufficient information may be available to make a conjecture with a higher degree of certainty.
- *Goals:* A programmer may find recurring computational code blocks, such as sort, search, delete, connect (to servers), start timers, transmit, and receive. It is useful to know the meaning of those recurring computational goals in the program, especially, why those computations are being performed, to be able to create a higher level of abstraction.
- *Problem domain:* Programmers need to have a broad understanding of the application domain of the code. A good understanding of the application domain serves as a backdrop for clearly and quickly understanding code segments in order to identify their goals and creating abstractions. For example, a programmer with a background in computer networks and distributed computing will find it easier to understand code for web-based applications than understanding the MATLAB libraries.
- *Plans:* Programmers have their own ways (also called plans) of finding solutions to problems that they need to solve. In addition, they use widely used solutions to some common problems, such as running-total loop, converging-iteration loop, handling a received message, and handling an event. Programmers' own plans and widely used plans by others are helpful in recognizing code patterns.

- *Rules of discourse:* Programmers have knowledge of *stylistic* conventions in writing code, which assist them in recognizing the goals of procedures and interpreting variables. For example, if a constant is called `MAX_RECORDS` and is used in a record processing loop, the programmer quickly recognizes that the loop is going to iterate for a maximum count of `MAX_RECORDS`. Similarly, if a method in Java is called `OpenTCPConnection()`, then the programmer recognizes that the program will most likely open a TCP connection with a remote computer. It takes much effort to understand variable names, such as `i`, with no clear meaning. One may guess that the variable is being used as a loop counter, an array index, and so on.

2. **External Representations:** The external representations of a program include its source code, documentation in the form of some comments, and manuals. The comments may appear in varying degrees of details, accuracies, and consistencies. The manuals are useful in understanding the high level goals of the code, whereas the inline comments are useful in understanding the low level details.
3. **Mental Representation:** By reading code and the accompanying documents, if there are any, ideally, a programmer may strive to create the following:
   - *Specification of the program:* Here specification means a complete and unambiguous description of the goals of the programs. The goals of the program can be specified by identifying the user-level functions (i.e., the functional requirements), attributes of the functions (i.e., the non-functional requirements), and constraints. Understanding the complete non-functional requirements and program constraints is more difficult than understanding the functional requirements.
   - *High level implementation of the program:* This means producing a complete and unambiguous description of the actions and data structures of the program. The actions can be identified at many levels of abstractions, such as individual statements, individual function calls, and sequences of calls. Similarly, the data structures can be identified at many levels of abstractions, such as individual variables, fields of structures (or, records), complete structures, and files. It is important to choose the right level of abstractions, because there are large numbers of individual statements and sequences of calls. Therefore, function calls appear to be the right kind of abstraction to describe actions.
   - *Annotation of the program:* Having understood the *goals* of the programs and identified the program's *actions* and *data structures*, it is important to make a two-way association between the goals and the *actions* and *data structures*. The two-way association is made by annotating the program in the following ways: (i) How each goal in the specification is accomplished and by which actions and data structures; and (ii) what goals use the services of a given action or a data structure. In other words, a programmer establishes a kind of traceability matrix between program goals and actions and data structures. If one views the development process of the program as a layered system

where the top layer is the set of program goals and the bottom layer is source code, then the annotation can be viewed as the abstraction of the middle layer connecting the top and the bottom layers. Basically, a program annotation is used to bridge the abstraction gap between the top and the bottom layers.

A programmer may not be able to create the final mental model, that is correct and complete, in one pass. The initial mental model may be incomplete and inconsistent. For example, many goals might be missing from the initial mental model. Similarly, the programmer might make mistakes in annotating the goals and the actions. Ambiguity may arise by identifying multiple goals with the same action (i.e., code segment).

4. **Assimilation Process:** Programmers combine their knowledge base and the external representations (i.e., documentation, code, and manuals) to create their mental models. Such a task is performed with a process called the *assimilation process*. The assimilation process guides the programmer to look at certain pieces of information, such as a code segment or a comment, and move forward/backward while reading the code. The assimilation process can work in three ways: *top-down*, *bottom-up*, and *opportunistic*. In the top-down approach, a programmer first identifies the goals of the program, followed by possible implementations of those goals such that the implementations match against the code. In the bottom-up approach, a programmer first identifies program plans from source code, makes annotations, and moves up to the top, goal layer. In reality, programmers rarely follow such pure top-down or bottom-up approaches. Rather, programmers combine the two approaches to take best advantage of whatever opportunity is available to make best progress in terms of knowledge gain at any given time. For example, one may start the assimilation process by reading code and identifying some goals. Next, the programmer may consider those goals and further understand the program in a top-down manner to gain a more complete knowledge about the identified goals and their implementations.
5. **Dangling Purpose Unit:** It may be recalled that the annotation layer serves as a *link* between the high level goal layer and the low level implementation layer, thereby showing what actions implement what goals and what goals have been implemented by what actions. Sometimes, it may be difficult to achieve a complete understanding of the relationship between the specification layer (i.e., the goals) and the low level actions. Therefore, the *dangling purpose* unit captures those goals whose implementations have not been clearly understood.

8.3.2 Shneiderman and Mayer Model

Shneiderman–Mayer's program comprehension model [9] has been depicted in Figure 8.3. The model consists of three key components: *short-term memory* of the programmer, the programmer's *knowledge* (both semantic knowledge and syntactic

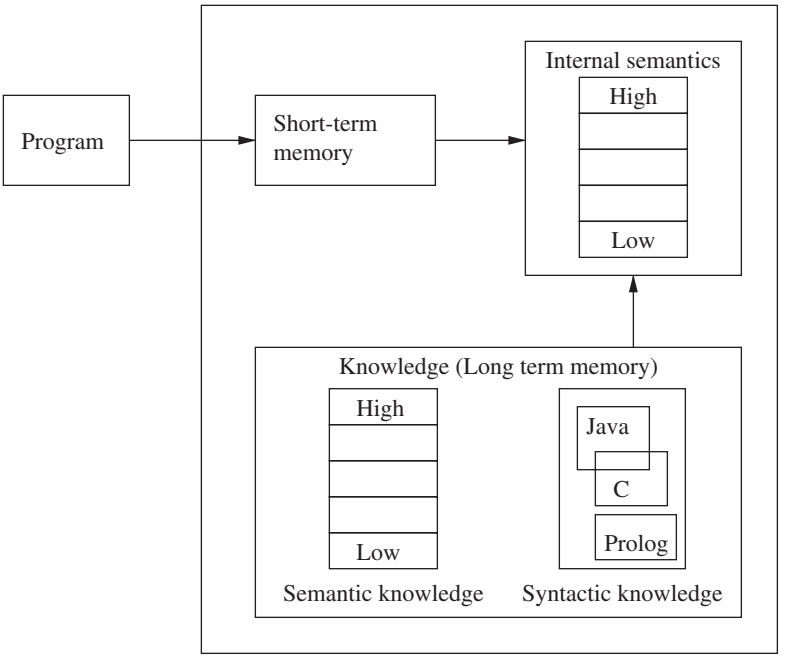

FIGURE 8.3 Shneiderman and Mayer program comprehension model

knowledge) to understand the code, and *internal semantics* of the code as understood by the programmer. In the following, we explain those three items in detail:

- *Internal semantics:* A programmer can read the code of a software system and develop an understanding of the code at different levels of abstractions. At the highest level of abstraction, the programmer develops an understanding of *what* the program does. In other words, develop an understanding of the functionalities of the program. The *what* aspect of a program gives us an idea about the *purpose* of the program. Some examples are as follows: a program inverts a matrix, a program encrypts a file with the triple data encryption standard (3DES), and a program sorts input records. One may understand the purpose of the program before understanding the statement-level details. At the lowest level of abstraction, the programmer understands the purpose of code segments and algorithms used to realize the program goals. In between the two extreme levels, programmers develop an internal semantic structure to represent the program details. An example of intermediate-level abstraction is the concept of *call graphs*—what modules are called by a given module and what modules use the services of the given module.
- *Knowledge:* Here knowledge refers to the application domain knowledge and programming knowledge, which are stored in the long-term memory of the programmer. Programming knowledge includes both semantic knowledge and

syntactic knowledge. On the one hand, semantic knowledge means programming concepts and techniques at different levels of abstractions. For example, (i) at a lower level, the programmer understands the meaning of individual program statements; (ii) at the intermediate level, the programmer understands how the values of two variables are swapped; (iii) at a high level, the programmer knows how to perform sorting and binary searching; and, (iv) at a further high level, a programmer with domain knowledge in computer networking knows how to route IP packets. Semantic knowledge is better gained and stored by means of abstractions, away from the details of programming languages. On the other hand, syntactic knowledge concerns individual programming languages and it is more specific, detailed, and generally arbitrary. For example, the concept of *value assignment* is represented by means of at least two notations in computer programming: "=" and ":=". Syntactic knowledge is quickly gained and quickly forgotten, whereas it takes more time to gain semantic knowledge and it stays longer in the long-term memory of programmers. Often languages share syntactic representations (Java and C languages share some syntactic structures.) The "knowledge" that we refer to in this section exists in the memory of the programmers.

- *Short-term memory:* While a programmer is reading code, *information* about the code enters the short-term memory of the programmer. The capacity of the short-term memory is very limited, and, therefore, the programmer must be able to quickly identify chunks, create their abstractions, and represent those abstractions in some internal form explained before. The programmer processes its short-term memory by using the semantic and syntactic knowledge of the code.

In the Shneiderman and Mayer model, program comprehension involves utilizing one's knowledge in identifying chunks of source code and creating an internal representation of the code in the form of a hierarchy of abstractions. One may have to create the hierarchy in several iterations, because it may not be possible to completely understand code in one pass of reading.

8.3.3 Brooks Model

There are three key elements in the program comprehension model of Brooks [10], illustrated in Figure 8.4 and explained in the following:

- *Code viewed as performing mappings from a problem domain to the programming domain:* Developing a software system can be seen as performing a series of mappings from one domain to the next, starting from the problem domain and finishing in the programming domain (i.e., solution domain). The results of the mappings are documented with varying degree of details, whereas the thought processes that perform the mappings are generally missing from the documentations. Let us consider the problem of routing data packets by a router on the Internet.

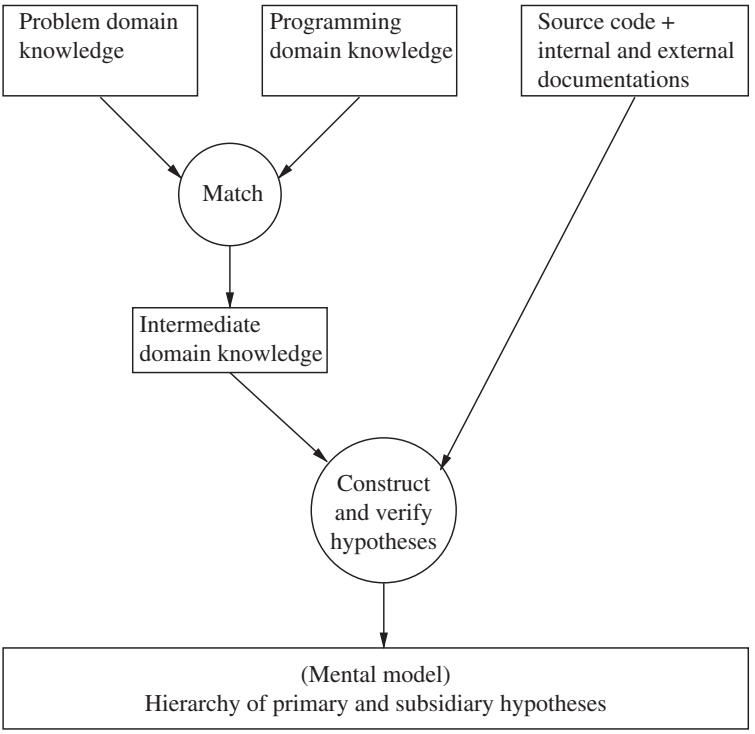

FIGURE 8.4 An overview of Brooks comprehension model

- *Step 1:* The input to a router is an IP data packet.
- *Step 2:* The *destination address* of the packet is *extracted* from the packet *header*.
- *Step 3:* A local *routing table* is *looked up* to find the *output port* on the router.
- *Step 4:* The IP packet is *transmitted* on that output port to reach the *next router* on the path to the *destination address*.

In Step 2, a standard data structure is used to extract the *destination address* value. In Step 3, a *prefix matching* algorithm is used for table look-up to find the *output port* to reach the destination. In Step 4, the IP packet is handed over to the lower protocol layer, namely, *data link* layer, with a request to transmit the packet on the given *output port*. Thus, the concept of packet routing has created a new level of domain knowledge, namely, standard data structure to access the fields of IP packets, prefix matching algorithm, and interfacing with data link layer. Next, the standard data structure for IP packets, prefix matching algorithm, and interfacing with data link layer are implemented in the programming language. Finally, when the router executes the code, an IP packet on a given input port appears on a certain output port of the router. This example illustrates the presence of three domains of knowledge: modeling of IP packets, modeling of

table look-up, and modeling of the interface with the lower layer protocol. One may note that there is domain-specific information in each domain and there is information about relationships between entities belonging to nearby domains. An example of relationship between entities in nearby domains is as follows: the model of an IP packet contains information about the destination address and the model of table look-up uses the destination address to find out the output port to reach the destination. For correct and complete understanding of the program, it is important to understand the mappings and their relationships, which is realized by means of constructing hypotheses and validating them.

- *Understanding the mappings in terms of hypotheses:* Programmers read all available documentations—requirements documents, source code, users manuals, maintenance manuals, and all other available information about the system—and try to reconstruct the mappings as much as they can. They are said to have truly comprehended the program if all the mappings are exactly reconstructed. However, for complex systems, such a task is far from complete and may be incorrect. Programmers try to understand the program by formulating hypotheses in terms of what they find from the available documentations, their expectations, their current level of understanding of the program, and their knowledge of the problem domain and programming in general. Hypothesis construction begins with the generation of a *primary* hypothesis concerning the global structure of the program in terms of inputs, outputs, major data structures, and the processing sequences. A programmer familiar with the program's application domain can form the primary hypothesis by looking at the name of the program and reading its brief description. The primary hypothesis must be general enough to cover the breadth of the program and specific enough to guide the construction of subsequent hypotheses. Hypotheses can be organized in a hierarchical manner to represent both the breadth and depth of comprehending the program. A large fraction of the hypotheses constructed by experienced programmers is likely to be valid, whereas novice programmers generally take more time in constructing valid hypotheses.
- *Verification and refinement of hypotheses:* A hypothesis represents a programmer's understanding of a certain aspect of the program— and that understanding may be correct, incorrect, or partially correct. Therefore, a hypothesis must be verified or refined by means of further understanding of the program. Programmers verify a hypothesis by searching the program text and related documentations for beacons that confirm the hypothesis. While reading code to find beacons, programmers try to develop a broad understanding of the program by having an open mind about the system, rather than stay focused only on the hypothesis under consideration. Therefore, strong beacons for any structure are very likely to be noticed, even if the structure is not related to the present hypothesis. Noticing strong beacons for structures, whether or not they are used in the verification of the current hypothesis, is useful for the following reasons:
  - The structure can be readily used in a later hypothesis, thereby speeding up program comprehension.

  - By discovering new structures in the code, a programmer can design new hypotheses. A new hypothesis can increase the breadth or depth of comprehension.

A programmer can continue constructing and validating hypothesis, thereby creating a hierarchical structure of hypotheses, where the top one is the primary hypothesis and the others are subsidiary hypotheses, and code segments are bound to specific hypothesis. Ideally, all program segments are unambiguously bound to the hypothesis structure, thereby indicating complete comprehension of the program. However, programmers may encounter a number of problems while verifying hypotheses.

- The programmer fails to find code to bind to a subsidiary hypothesis.
- The same code is bound to multiple subsidiary hypotheses.
- The programmer fails to bind a code segment to any hypothesis.

The above problems can be resolved by adopting new hypotheses, refining the existing hypotheses, and altering and adding to the bindings of code segments to hypotheses. Therefore, programmers need to iterate the process of hypothesis construction and verification. Brooks has identified a number of factors having an impact on program comprehension, as explained in the following:

- *Programs differ in comprehensibility:* We can identify two broad aspects of programs that affect their comprehensibility, namely, characteristics of the executable portion of the source code and the quality of documentation. In the following, we explain the above two items in detail:
  - *Characteristics of source code:* The number of identifiers, the number of statements, and the amount of branching affect program comprehension. If a programmer finds a large number of identifiers and conditional statements in a program, they need to keep a lot of information in their short-term memory and will ask many "why" questions while reading code. However, the constructs that have a higher level of impact on program understanding are the *control* constructs. For example, a program with many `for` and `while` loops make understanding difficult. Concurrency and synchronization also make program understanding very difficult. Finally, process synchronization techniques, such as synchronization via global variables, make program understanding very difficult.
  - *Quality of documentation:* Documentation of programs can be classified into two groups: *internal* and *external*. Internal documentation is closely intertwined with the program text, whereas external documentation are found in entities separated from source code. The programmers can find internal documentations in the following *indicator* forms:
    * prologue comments, including data and variable dictionaries;
    * variable, structure, procedure and label names;

- declarations;
- comments associated with statements and code segments;
- code indentation; and
- input/output formats and header.

On the other hand, external documentation includes the following *indicator* forms:

- user's manual;
- program logic manuals;
- flowcharts;
- cross-reference listing;
- published descriptions of algorithms; and
- published standards referenced by the program.

The usefulness of a certain documentation is a function of the beacon it describes and the importance of the beacon for the specific hypothesis under consideration. Different documentations are useful in different stages of hypothesis construction. For example, in the initial stages of hypothesis verification, the hypotheses tend to be very broad and focus on the global properties of the system. Therefore, documentations with a focus on the overall functionality of the system, such as user's manual and flowcharts, are more effective in comprehending the program. It is important to note that more documentation may not necessarily lead to better accuracy in program comprehension. If a program concept has been described by means of many different indicators, it is likely that the programmer will easily notice one form and use it. For instance, a functionality of a program can be considered as a program concept, and the concept may be described in different ways in different places: requirements specification, user's manual, and inline comments. As a concrete example, "draw a line" is a functionality of a graphics program, and this functionality can be described in many ways in many places. Different indicators mean different ways of describing a program concept. For example, the "draw a line" functionality is generally described in different ways in requirements specification, user's manual, and inline comments. On the other hand, if a large number of indicators redundantly present the same information, then they may obscure or overwhelm other indicators that contain unique information. In other words, over documentation may be counter-productive to program comprehension. In addition, a programmer needs to be aware of the possibility of contradicting indicators, that is, the user's manual says one thing and the inline comments say something contradicting.

- *Task differences affect comprehension:* Programmers may comprehend the same program from different viewpoints, depending upon the *tasks* they want to perform after comprehending the program. Some examples of tasks that programmers may perform on the program are: fixing bugs, modifying the contents and format details of output, and increasing the readability of code. A programmer

who needs to fix bugs in a program needs to be more concerned with the control structure of the program, whereas a programmer who is trying to reformat the output will be more focused on the output statements. Therefore, a programmer's perspective of the source code affects how they start their comprehension process.

- *Programmers differ in their ability to comprehend programs:* Programmers possess different abilities to comprehend programs. Their individual abilities to construct effective hypotheses and verify them depend on the programmer's following abilities:
  - Domain knowledge: A programmer's knowledge about the problem domain plays an important role in understanding the top level goals of the program and the mappings from layer to layer. For example, a programmer with deep mathematics background is better positioned to understand code for numerical analysis than a programmer without much mathematics background. Similarly, a programmer with background in aeronautics is likely to be more capable of quickly understanding code for airplane wing design.
  - Programming knowledge: It is needless to say that programmers must have a thorough knowledge of the programming language to understand the code details.
  - Comprehension strategies: Comprehension strategies include a large number of fine decisions: where to start, how to proceed, how to construct hypothesis, how to verify hypothesis, how to refine hypothesis, how quickly to reject hypothesis, when to postpone the verification of a hypothesis, when to reconsider a hypothesis whose verification had been postponed before, and how to read code and documentations. An experienced programmer may move between program text and documentations quickly, whereas a novice programmer may read everything line-by-line.

In summary, the comprehension model of Brooks puts emphasis on constructing a hierarchy of verified hypothesis and associating code segments with lower level, subsidiary hypotheses. A programmer is said to have completely and correctly comprehended a program if each code segment has been assigned to some hypothesis, no code segment remains unassigned, and no code segment is assigned to multiple hypothesis. An overview of the Brooks model has been illustrated in Figure 8.4.

8.3.4 Soloway, Adelson, and Ehrlich Model

The top-down program comprehension model of Soloway, Adelson, and Ehrlich [4, 11], depicted in Figure 8.5, works in a top-down manner, and it applies when the code or type of code is familiar to the programmer. Two fundamental concepts in the model are *programming plans* (also called *schemas*) and *programming rules of discourse*, as explained in Section 8.2.3. Expert programmers have and use specific programming plans and rules of programming discourse to comprehend programs.

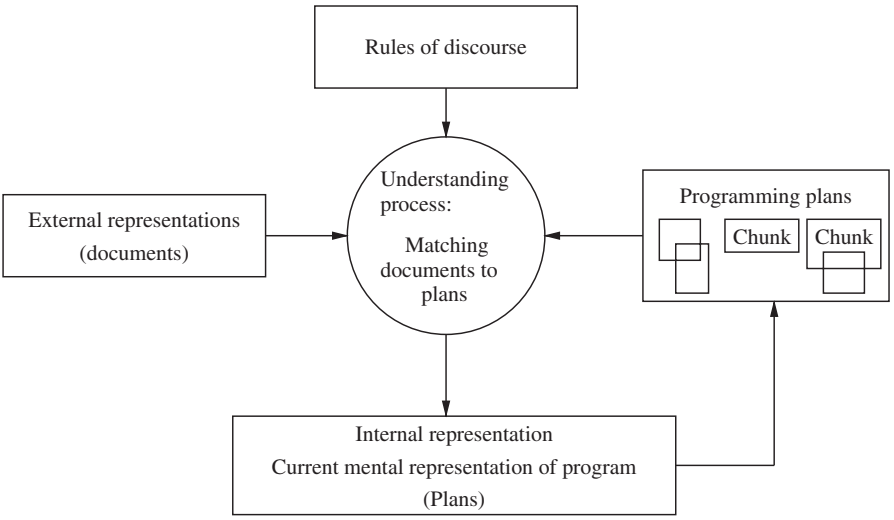

FIGURE 8.5 Soloway, Adelson, and Ehrlich comprehension model

Programs that do not follow the plans and violate the rules of discourse are harder to understand. Some concrete rules of programming discourse are as follows:

- The names of the variables reflect their purpose.
- Code that is not going to be executed is not included.
- A tested condition must have the potential of evaluating to *true*.
- A variable that is initialized by means of an assignment statement is subsequently updated with assignment statements.
- Use an `if` statement to execute a code segment once, whereas `for()` and `while()` loops are used to repeatedly execute code segments.

Expert programmers have strong expectations about what programs should look like. When those expectations are not fulfilled, their comprehension performance drops significantly, almost to the levels of novice programmers.

Similar to the other models, external representations, namely, documentations, play a key role in program understanding in this model. Examples of documentations are requirements documents, design documents, source code, user manuals, reference manuals, maintenance manuals, test case documents, test execution reports, and even field reports. As depicted in Figure 8.5, the understanding process matches programming plans found in source code with external documentations using rules of discourse. During the understanding process, the programmer creates a hierarchical knowledge structure representing their understanding of the code. Therefore, comprehension begins with a high level program goal, and finer, lower level subgoals are generated to realize the upper level goals. Programming comprehension is an

iterative process, such that, in each iteration, the programmer expands their understanding of the code by refining the already identified subgoals and identifying new subgoals. The process is said to be complete when the programmer has associated all the programming plans with the goal hierarchy.

8.3.5 Pennington Model

The Pennington model [12, 13] has been depicted in Figure 8.6. The model applies two concepts, namely, *textbase* and *situation model*, developed by van Dijk and Kintsch [14]. It is important to pay attention to the loop {*Match – Mental representation – Text structure knowledge*} followed by the *textbase*. The programmer iterates through the loop, thereby incrementally creating the mental representation. Finally, when the programmer stops iterating through the loop, the final mental representation is known as the textbase. There is a similar relationship between the second *mental*

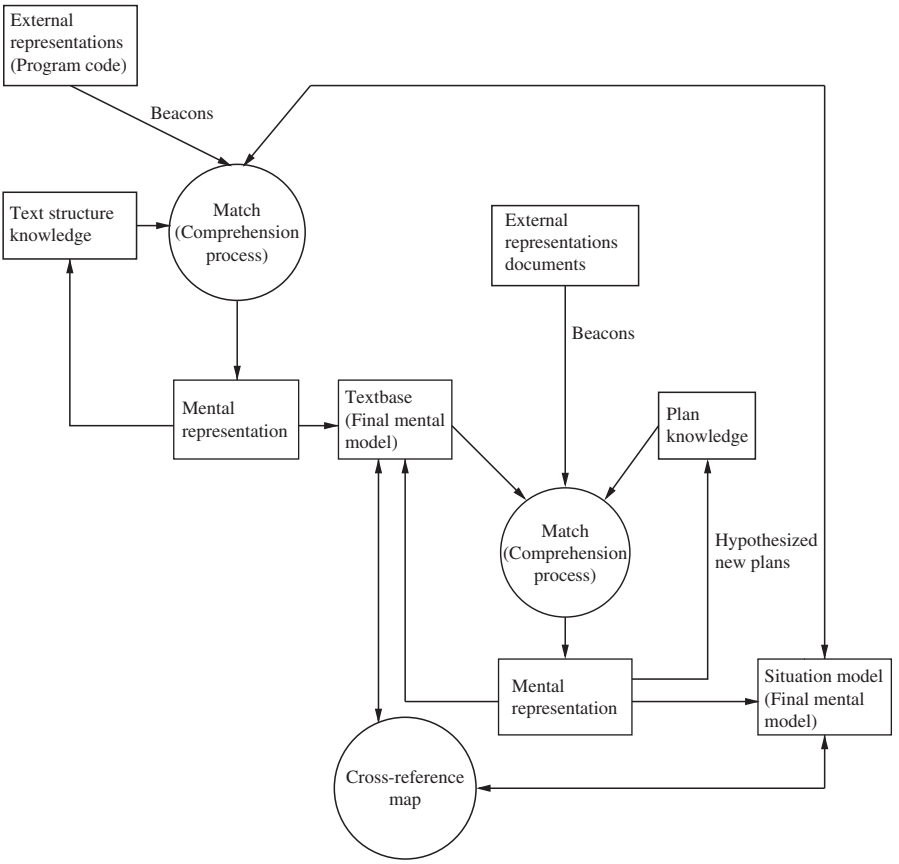

FIGURE 8.6 Pennington model

representation box and the *situation model.* The concepts of *textbase* and *situation model* are explained as follows.

- *Textbase (Program model):* Some people equate being able to recall portions of a text with comprehending the text. A textbase represents information that the reader of a text can recall from memory after reading the text. A textbase includes a hierarchy of representations comprising a surface-level knowledge of the text, a *microstructure* of relationships among text propositions, and a *macrostructure* organizing the text representation. One may note that a textbase does not necessarily contain the exact words or lines used in the text. Though a textbase represents the superficial understanding of a programmer, it is a useful abstraction of a program for many reasons, because major code segments are recognized more easily than minor segments. The textbase basically describes a *program model* in terms of the control flow of the program, because when programmers read new code they build a control flow abstraction of the code. This model is built in a bottom-up manner by:
  - identifying beacons for elementary code blocks;
  - chunking microstructures into macrostructures; and
  - cross-referencing the existing contents of the textbase with the situation model.
- *Situation model:* A situation model represents *what the text is about*. The model requires knowledge of the real-world domains and objects. Programmers mentally represent the code in terms of real-world objects by using plan knowledge. For example, the code line "`status = delete(ProcessQueue, printJob1);`" is mentally represented as "delete the `printJob1` from the queue of processes." Also, lower level plan knowledge is chunked into higher level plan knowledge by using the concepts of operations, control flow, data flow, state, and functions. The situation models are built via cross-referencing and chunking. To build the situation model, the cross-referencing process takes information from the textbase and builds higher order plans by means of constructing hypotheses.

A high level description of the model is as follows:

- The programmer assimilates their understanding of the code, knowledge of the text structure, and the situation model to create a mental model in the form of a textbase.
- The programmer assimilates the external representations of the code, (i.e., documentations), plan knowledge (i.e., problem domain knowledge and intermediate-level programming concepts), and the textbase to create the situation model.
- The textbase and the situation model are cross-referenced to refine and update the two models.

Figure 8.6 indicates that the program model represented by the textbase can change after a programmer starts constructing the situation model. In addition, a new program

model can impact the situation model. Cross-referencing between the program model and the situation model enables the programmer to maintain consistency between the two models.

While reading code, programmers gain knowledge about the following aspects of code to varying degree of details.

- *Operations:* At the level of source code, operations are *actions* that the program performs. Some examples are assigning a value to an integer variable, comparing the values of two integer variables, and swapping the values of two character variables. Therefore, operations can have varying levels of granularities. After reading code for a while, a programmer learns what operations the program is performing.
- *Control Flow:* Control flow refers to the ordering of operations (i.e., actions) in a program. By looking at sequences of actions, a programmer can identify *before* and *after* relationships between actions. For example, a programmer can ask: *Has a timer been started after transmitting a data packet?* and *What actions are taken after a timer expires?*
- *Data Flow:* Data flow refers to the sequences of transformations being performed on input data to produce program outputs. The concept of input data is very broad, including values of variables and file contents. Variables are defined (i.e., initialized), used, redefined, reused, and so on. The value of a variable can be used in two ways: (i) to perform a computation, known as c-use; and (ii) to construct a predicate, known as p-use. For example, a variable appearing in the condition portion of an `if()` statement is an example of predicate use of the variable. By understanding the data flow aspect of a program, one can ask: *What are the input variables that determine the values of output variable* `LossCount`*?*
- *State:* In general, the values of all variables of a program define the state of execution of the program. Programmers can ask useful questions relating to program states: *When an acknowledgement packet arrives, has the timer expired?*
- *Function:* Programs can be characterized by top level goals and subgoals. Also, programs can be characterized by hierarchies of functions performed by code blocks. Functional relations concern the goals and subgoals of the program and their associations with code segments. By understanding functional relations in a program, one can answer questions such as *Is this loop computing the average monthly credit of a customer?* and *Is this sequence of actions establishing a phone connection between two customers?*

8.3.6 Integrated Metamodel

When programmers read the code and documentations, they start understanding the code at all levels of abstractions simultaneously. As they read the code, they simultaneously understand the program's control flow (i.e., *how* the program works), the program's functions (i.e., *what* the program does), and the overall situation of the program (i.e., what the code *is about*). Moreover, programmers do not follow a

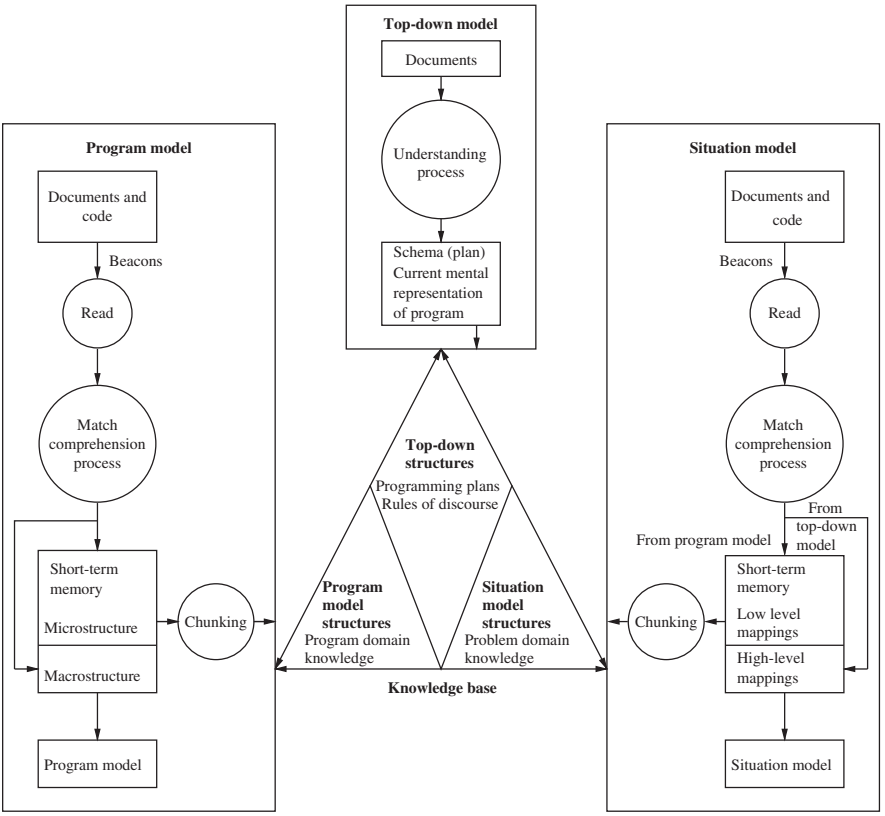

FIGURE 8.7 Integrated Metamodel. From Reference 1. © 1995 IEEE

purely top-down or bottom-up approach to comprehending code. Rather, they become opportunistic and their strategy combines both top-down and bottom-up approaches.

The integrated code comprehension metamodel proposed by Mayrhauser and Vans [1, 15] has been depicted in Figure 8.7. The four major components of the model are explained in the following.

- *Top-down model:* The top-down model, also known as *domain model*, represents domain knowledge about the program [11]. For example, domain knowledge about IP routing includes IP packet header format, IP table look-up, prefix matching, and packet transmission. Similarly, domain knowledge about operating systems includes process scheduling, memory management, and device management. Domain knowledge provides guidance to programmers while designing effective strategies to read and understand code. Construction of the top-down model is effective for familiar code or type of code, because it requires that the reader possess domain knowledge.
- *Program model:* When a programmer reads unfamiliar code, he tries to understand the *control flow* aspect of the code via his programming skills and standard

programming constructs [12, 13]. If the programmer understands *what* the program is doing from the control flow aspect, then he has a *program model* of the code. However, the programmer has not developed a complete understanding of the functions of the program.

- *Situation model:* We have already explained the concept of a situation model while discussing Pennington's model in this section. After building a program model (or, at least a portion of the program model), a programmer builds a situation model by using control flow and data flow information in a bottom-up manner. Referring to the earlier example of IP routing (see the top-down model above), *packet forwarding* is an example of the situation model.
- *Knowledge base:* The knowledge base provides a medium for interactions among the above three models, namely, top-down model, program model, and situation model, and comprises the following three kinds of knowledge structures:
  - *Top-down structures:* The top-down structures include two key elements from the Soloway, Adelson, and Ehrlich model, namely, *programming plans* and *rules of discourse*. The programming plans are categorized into *strategic plans*, *tactical plans*, and *implementation plans*. Strategic plans describe language-independent, global strategies used in a program. An example of a strategic plan is the state model of packet discard policy used by an IP router. Tactical plans are local strategies for solving problems described by the higher level strategic plans. Therefore, strategic plans are further decomposed into language-independent tactical plans. For IP routing, tactical plans may include classification of packets based on the time elapsed so far since their creation, the average times needed to route them to their destinations, the service category of the packet (e.g., voice-over-IP). Algorithms for management of packet queues are also a part of tactical plans. Strategic plans are understood by means of fine-grained tactical plans. Finally, implementation plans are language dependent and are used to code tactical plans. A Java method used to implement a queue to reflect the order of arrivals of packets is an example of implementation plan. Such a queue can be implemented as a doubly linked list.
  - *Program model structures:* The program model structures include two kinds of knowledge, namely, *text-structure knowledge* and *plan knowledge*. Text-structure knowledge is described by means of control primes, and plan knowledge is described in terms of algorithms, control flow, data flow, and data structures.
  - *Situation model structures:* The situation model structures are described in terms of *problem domain knowledge* and *functional knowledge*. The problem domain knowledge is also known as *real-world knowledge*.

In the integrated metamodel, all the three models, namely, the top-down model, the program, and the situation model, are simultaneously built. Here, the meaning of simultaneity is that no model waits for another model to be completely built; rather, the programmer takes an opportunistic approach and simultaneously built all the three models by updating the knowledge base with the understanding of one model and

applying the knowledge to further build another model. For example, the knowledge generated from the program model can be used to expand and/or refine the situation model, and vice versa. The programmer uses the knowledge base for guidance to make a transition from one model to another.

The knowledge base is also called *long-term memory*, and it is generally organized into *schemas* (or, *plans*) as explained before in this section. Schemas are knowledge structures with two parts: *slot types* and *slot fillers*. Knowledge structures are linked by means of the commonly used concepts of modeling associations, namely, *kind-of* and *is-a* relationships. As shown in Figure 8.7, the knowledge base is partitioned into three groups specifically related to the three comprehension models: *programming plans* and *rules of discourse* are related to the *top-down* model; *program domain knowledge* is related to the *program model*, and *problem domain knowledge* and *functional knowledge* are related to the *situation model*. However, structures built by any of the three models are available to the other models.

8.4 PROTOCOL ANALYSIS

Novice programmers can learn by observing how experienced programmers *behave* during program comprehension. Similarly, researchers can learn by observing how both novice and experienced programmers behave during program comprehension. Ideally, we want to observe all aspects of programmers' behavior while they are trying to understand the code, such as the following:

- What is the programmer studying?
- What is the programmer thinking when he sees something interesting?
- What does the programmer do after he finds something interesting?
- What is the rational thinking behind the programmer's action?

Answering the above questions is a cumbersome—if not impossible—task. An elaborate mechanism and much resources are required to be able to observe how programmers behave during program comprehension. *Protocol analysis*, studied in the field of psychological research, is a key concept used in finding answers to the above questions. It is a methodology for eliciting verbal reports from participants (programmers in our case) about their thought sequences as a valid source of data on thinking [16]. In the field of psychology, protocol analysis is used to study cognitive processes and behavior analysis. It has also been used in usability testing and educational psychology. Protocol analysis is composed of two sequential steps:

- *Concurrent verbalization of a comprehension task:* This step produces textual data representing the thought sequence of a programmer as he performs the comprehension task by reading the code. This text is known as *protocol data*.
- *Analysis of protocol data:* The protocol data is analyzed to understand the characteristics of the thinking performed by the programmer.

Concurrent verbalization of comprehension tasks: Programmers trying to understand a software are asked to verbalize their thoughts while working on a specific task, and the said verbalization is recorded on audio-visual systems. In other words, programmers are asked to "think aloud" while working: they say loudly everything that they think, evaluate, and (mentally) move. This is called *think-aloud protocol* (TAP). Thinking aloud reflects the knowledge currently active in the programmer's working memory. The verbal reports almost always indicate the programmer's goals, operators, and evaluative processes [17]. Some examples of concurrent verbalization is as follows:

- I want to read the external documentations. What? No external documentations! I wanted to speak with the developers who designed and implemented the system, but they are all gone! I mean they have left the company.
- Okay. I am reading the code prologue.
- Now I know that the system is for enabling customers to make seat reservations in a restaurant and placing orders.
- I find interesting keywords in the prologue: phone, cell phone, and laptops. I think one could make reservations by calling a restaurant or on the Web. I guess … customers might be able to place orders from their cell phones.
- Let me read the module called `MakeReservation`.
- …
- On line# 206, I see `MobileDeviceType` in a data structure to interpret received messages from clients. This verifies my guess that the system supports user interactions via smartphones.
- …

The above thinking-aloud action of the programmer can be recorded and transcribed. The transcribed verbal report is called *protocol data* (or, simply *protocol*) about the comprehension process. Protocol data is a list of statements, questions, and, sometimes, amazements, uttered by a programmer while reading code and documentations. Protocol data reflects not only the programmer's thought process, but also much interesting details noticed about the program. Different programmers with the same level of experience will most likely produce different protocol data for the same program, because they are likely to think differently. Protocol data produced by a novice programmer and an experienced programmer will be very different in their details, sources of those data, and reasoning.

Analysis of protocol data: There is no common, detailed procedure to analyze protocol data. Rather, a very general description of protocol analysis is as follows.

- Divide protocol data into several *segments*, say, *speech sentences*.
- Assign the segments to different *predefined categories*. Assignment of protocol segments to different categories is called *encoding*. Coding categories are generally selected with a model of the verbalization process in mind. For example, in the coding system of Ericsson and Simon [16], there are four types of

segments: *intentions*, *cognitions*, *planning*, and *evaluations*. Intentions are easily recognized by verbs of the type, "shall," "will," "must," and so on. The cognition category contains segments which put attention on *selected aspects* of the current situation, and those segments are recognized by constructions indicating *presence* and *immediacy*. Planning is easily recognized by conditional constructions, namely, "if A then B else C." Evaluations are recognized by indications of implicit and explicit comparisons of alternatives, namely, "yes," "no," "okay," and "fine."

- Analyze the categorized protocol data to build a comprehension model of the programmer. A comprehension model can be represented as a *transition net*, which resembles finite-state machines, where computations (represented with cognitions and intentions) are associated with states, and planning and evaluations are associated with transitions. Transition nets are easy to construct and understand, and those are simple tools to model thought processes, verbalized as a TAP, of programmers.

8.5 VISUALIZATION FOR COMPREHENSION

Software visualization is one of the techniques applied in reverse engineering, and it has been explained in detail in Section 4.6.6. In this section, we further explain the idea as it has been supported with various tools for program comprehension.

Cleveland [18] has described a tool, called Program UNderstanding Support environment (PUNS), developed at IBM to provide *multiple views* of a program to aid its understanding. The tool supports the program understanding task by organizing and presenting the different viewpoints of a program: a *call graph* for a set of procedures, a *control flow* graph for an individual procedure, a graph representing the relationship between a file and a procedure that uses the file, a *data flow* graph, and a *definition-use* chain for a variable. By performing static analysis of the code, the tool detects low level relationships and organizes them in a user-friendly environment so that the user can easily navigate through the graphs while switching between low level and high level objects. The two key components of the tool are as follows: (i) a repository and the associated routines—the routines load the repository and generate responses to queries from the user; and (ii) a user interface presents information about the program to the user and aids the user in navigating through the graphs.

Harandi and Ning [19] have presented a program analysis tool (PAT) to represent programming concepts with an object-oriented framework and a heuristic-based concept recognition mechanism to extract high level functional concepts from source code. The tool is based on the human expert's analysis model, and tries to aid programmers answer the following questions. (i) What high level concepts does the program implement? (ii) How are the high level concepts coded in terms of low level details? Two types of knowledge are explicitly represented by PAT: *program knowledge* and *analysis knowledge*. The former is represented by programming concepts found in the code, whereas the later is represented by information contained in program plans. The tool manages two databases to manipulate the two types of knowledge: (i) a database

of coding heuristics, data structure definitions, and functional coding pattern; and (ii) a database of rules for program plans covering value accumulation, counting, sequential search of ordered and unordered structures, different kinds of searching, and sorting.

The positive impact of *software visualization*, by means of *control structure diagrams* (CSD), on program comprehension has been shown by Hendrix, Cross II, and Maghsoodloo [20]. Graphical notations are used to render source code in the CSD approach. The CSD is an algorithmic level diagram that clearly depicts control constructs, control paths, and the overall structure of each program unit, say, function or method. The CSD is a kind of extension to other diagrams, such as class diagrams and data flow diagrams. The authors performed experiments with two groups of students: one group was given the source code in plain text only, whereas the second group received the source code rendered with CSD. All the students were asked to respond to questions regarding the structure and execution of one source code module of a public domain graphics library. The response time and the correctness of the responses were recorded. Statistical analysis revealed that the CSD significantly enhanced the students' program comprehension capabilities.

The concept of *fisheye* view [21] is another visualization technique to support programmer's navigation and comprehension. A fisheye view displays those parts of the source code that have the highest degree of interest (DOI) related to the current focus of the programmer. Jakobsen and Hornbaek [22] have presented the design of a fisheye interface based on the Eclipse Java development environment. Both the proposed and Eclipse's interfaces use an *overview plus detail* approach: an overview of the entire document is displayed to the right of the detailed view window. The overview displays the source code reduced in size to fit the entire document within the space of the overview area. Because of size reduction, the text displayed in the overview area is unreadable, but one can discern some structural features of the code, such as method boundaries and blocks of comments. The portion of the code shown in the detail area is visually connected with its location in the overview. In addition, the fisheye interface of Jakobsen and Hornbaek possesses the following features.

- *Focus and context area:* The detailed view of the code divided into two areas: the focus area and the context area. The context area displays the code context of the code shown in the focus area. Moreover, the context area displays a distorted view of the context, because much code needs to be squeezed into a small area – and this is achieved by diminishing the font size of text.
- *Degree of Interest Function:* A DOI function is used to determine if and how much the text lines are diminished in size in the context area. The DOI function is based on an a priori interest defined by: (i) the type of code line for which the DOI is being calculated; and (ii) the line's distance from the focus point. The type of a code line is determined by deducing the abstract syntax tree node from the line. Code lines containing keywords `package`, `class`, `interface`, or `method` are assigned a higher a priori interest than other lines. Lines containing `if`, `for`, `switch`, `try`, `while`, `catch`, or `finally` are also assigned a higher a priori interest. The distance of a line from the focus point is calculated by using

the concepts of *syntactic distance* and *semantic distance*. Syntactic distance is measured in terms of the "gap" in their indentation levels; that is, lines in the same indented block as the focus point are thought to be closer to the focus point than lines on other indentation levels and lines in other blocks, thus contributing to a higher DOI. Semantic distance is about the similarity of information contained between two lines; that is, lines containing declarations of classes, methods, and variables that are used in the focus point are considered to be closer to the focus point, thus contributing to a higher DOI. Lines with semantic closeness to the focus point are highlighted with an alternate background color to show their semantic relation to the lines in the focus point.

- *Magnification Function:* A magnification function assigns a priority to each line of code according to its DOI so that the font size of less interesting lines can be reduced. Lines with similar degrees of interest are assigned priorities according to their distance in lines from the focus area so that lines closest to the focus area are allocated space first. Such a strategy is useful in deciding how to effectively use the display space.
- *User Interaction:* The focus area offers the same functionalities for interactions with the code file as a standard text editor. As the user navigates through the focus area, the context area automatically changes. Conversely, clicking on a line in the context area highlights the focus area around that line.

The authors conducted an experiment with 16 participants to compare the usability of fisheye view with a common, linear presentation of code. The participants generally preferred the fisheye view interface and performed comprehension tasks significantly faster.

UML (Unified Modeling Language) class diagrams can aid programmers in code comprehension, and the spatial organization of those diagrams play a key role [23]. There are numerous algorithms for drawing UML class diagrams (see [23]). Based on *perceptual theory* [24], Sun and Wong [23] have classified criteria and guidelines for effective layout of UML class diagrams. Several theories of perception have been proposed: *Marr's theory*, *Gibson's theory*, *Gestalt's theory*, and *Theory of notation*. Two broad principles, namely, *perceptual organization* and *perceptual segregation* can be applied to organize UML class diagrams. The former indicates when entities are organized in near proximity, whereas the latter indicates when entities are separated in Gestalt's theory of figure–ground segregation. The following are some important principles of *perceptual organization*, most of which are from Gestalt's theory.

- *Good figure:* Generally, simpler figures with the right amount of details are considered to be good. Complex figures and figures with a lot of details do not increase our understanding of structures represented by those figures.
- *Similarity:* Similar entities appear to be grouped together. For example, entities with the same shape, say, a circle, appear to indicate the same concept, say, processing.
- *Proximity:* Entities that are close to each other are grouped together.

- *Familiarity (Meaningfulness):* If a group of elements appear to be familiar, it is more likely that they are grouped together.
- *Element Connectedness:* Entities that are physically connected are seen to be constituting a single unit.

On the other hand, the concept of perceptual segregation is explained as follows. When one looks at the environment, what is seen is a whole picture, not as separate parts. There are images in the environment that people are aware of in a particular moment—this would be the figure. The ground makes up the image if one does not focus on it or one is not aware of it. The following factors make an entity more like a figure that can be easily recognized.

- *Symmetry:* Symmetric areas are commonly seen as a figure.
- *Orientation:* Entities with horizontal and vertical orientations are more likely to be seen as figures than other orientations.
- *Contours:* Edges are useful in perceiving if something is a figure.

The above explained principles of perceptual organization and perceptual segregation are expressed in terms of "laws" for drawing UML class diagrams. Those laws are called:

- Law of good figure
- Law of similarity
- Law of continuation
- Law of proximity
- Law of familiarity
- Law of connectedness
- Law of symmetry
- Law of orientation
- Law of contour.

For example, the law of continuation is described as: *minimize edge crossings and bends*, because it is easy to follow a class diagram with minimum number of edge crossings and bends. They have evaluated the class diagrams produced by two tools, namely, *Borland Together* and *Rational Rose*, by applying the above laws.

Wettel and Lanza [25] have proposed a three-dimensional (3D) visualization of programs based on the *city metaphor*. With the city metaphor, classes are represented as buildings located in city districts which in turn represent packages. The concept of *habitability*, which makes a place livable, is at the core of the city metaphor, and the corresponding programming concept is *familiarity*. The more familiar a programmer is with the code, the easier it is to understand the code. Thus, intuitively, habituality is the characteristic of code that enables programmers coming to the code later in the software system's life to understand its intentions and constructions [26]. They

also support the concept of *locality* by providing a navigable environment in which one can move. In summary, the programmer perceives code as a city with visual orientation points and freedom of movement and interaction.

8.6 SUMMARY

The chapter began with a general introduction to program comprehension. Next we explained the concepts of *knowledge* and *mental model*. The concept of mental model was explained by means of the static elements of mental models, dynamic elements of mental models, and acquiring knowledge from code. This was followed by a discussion of the characteristics of expert programmers. Static elements of mental models include: text-structures, code chunks, schemas, plans, and hypotheses. On the other hand, the dynamic elements of mental models are: chunking, cross-referencing, and strategies. Next, we explained the following comprehension models: Letovsky model, Shneiderman and Mayer model, Brooks model, Soloway, Adelson, and Ehrlich model, Pennington model, and Integrated metamodel. Studying programmers' behaviors and actions during their program comprehension processes are key to understanding the details of their thinking. Therefore, we explained the concept of *protocol analysis* that involves the *think-aloud protocol* in which programmers are asked to verbalize their thinking. Finally, we explained the idea of program understanding by means of software visualization.

LITERATURE REVIEW

A programmer is said to have understood a program when he is able to explain the program by means of human-oriented *concepts* different from the *tokens* used to construct the source code of the program. Examples of human-oriented concepts are the program's structure and behavior and the program's relationships to its application domain. The concept "Allow the customer to withdraw money" is a high-level concept, which is much different from the following C-program statement:

```
if (Balance > RequestedAmount && DailyLimitNotReached)
      then GrantWithdrawalRequest(RequestedAmount, Customer).
```

While reading code, it is important that programmers are able to associate high level concepts to code segments. Biggerstaff, Mitbander, and Webster [27, 28] have studied the *concept assignment problem*: discovering the human-oriented concepts and assigning them to their implementations as code segments. Human-oriented concept recognition requires plausible reasoning and relies on a priori knowledge from the specific application domains. Two general steps are applied when attempting to assign concepts to code: (i) identify which entities and relationships in a large program are important; (ii) assign those entities and relationships to known or discovered domain concepts and relations. Deep understanding of programs relies on

an a priori knowledge of the expectations about the problem domain and commonly used software architectures. The authors have reported a *desi*gn *re*covery tool, called DESIRE, developed at the Microelectronics and Computer Consortium. The tool uses informal knowledge, such as variable names and comments, a domain model, and traditional formal analysis to build a hierarchy of concepts to collectively describe a program.

Gold and Bennett [29] have proposed a hypothesis-based concept assignment method. The method assigns *descriptive terms* to their implementations in source code. The terms relate to computational intents and are nominated by the programmer. Informal information, such as identifier names and comments are used in assigning concepts. The method works in three phases: (i) generate an ordered list of hypotheses from code; (ii) group the hypotheses using their conceptual affiliation to produce a list of hypothesis segments; and (iii) evaluate the evidence in the segments and assign a concept to each segment. The overall output of the method is a list of concepts, associated with blocks of source code. The effectiveness of the method has been demonstrated by applying it to legacy code in COLBOL II.

Tonella [30] has proposed to represent a program as a *concept lattice of decomposition slices*, by combining the concepts of *decomposition slice graphs* and *concept lattice*. A decomposition slice graph is obtained by applying the idea of program slicing [31]. Visual inspection of this representation aids programmers in understanding the decomposition of the program into functionalities and subfunctionalities.

REFERENCES

[1] A. V. Mayrhauser and A. M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer*, 28(8), 44–55.

[2] P. N. Johnson-Laird. 1983. *Mental Models*. Harvard University Press, Cambridge, MA.

[3] A. C. Graesser. 1981. *Prose Comprehension Beyond the Word*. New York: Springer-Verlag.

[4] E. Soloway and K. Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 595–609.

[5] S. Letovsky, 1986. *Cognitive Processes in Program Comprehension*. Proceedings of the First Workshop in Empirical Studies of Programmers, 1986, Ablex, Norwood, NJ, pp. 58–79.

[6] I. Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23, 459–494.

[7] R. Guindon. 1990. Knowledge exploited by experts during software systems design. *International Journal of Man-Machine Studies*, 33, 279–282.

[8] R. Jeffries, A. A. Turner, P. Polson, and M. E. Atwood. 1981. The processes involved in designing software. In: *Cognitive Skills and Their Acquisition* (Ed. R. R. Anderson) , Erlbaum, Hillsdale, NJ.

[9] B. Shneiderman and R. Mayer. 1995. Syntactic and semantic interactions in programmer: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3), 219–238.

[10] R. Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 543–554.

[11] E. Soloway, B. Adelson, and K. Ehrlich. 1988. Knowledge and Processes in the Comprehension of Computer Program (Eds M. Chi, R. Glaser, and M. Farr), pp. 129–152. A. Lawrence Erlbaum Associates, Hillsdale, NJ.

[12] N. Pennington. 1987. *Comprehension Strategies in Programming*. Proceedings of the 2nd Workshop on Empirical Studies of Programmers, 1987, Ablex, Norwood, NJ, pp. 100–112.

[13] N. Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19, 295–341.

[14] T. A. van Dijk and W. Kintsch. 1983. *Strategies of Discourse Comprehension*. Academic Press, New York.

[15] A. V. Mayrhauser and A. M. Vans. 1994. *Comprehension Processes During Large-scale Maintenance*. Proceedings of the 16th International Conference on Software Engineering, 1994, IEEE Computer Society Press, Los Alamitos, CA, pp. 39–48.

[16] K. A. Ericsson and H. A. Simon. 1984. *Protocol Analysis: Verbal Reports as Data*. MIT Press, MA.

[17] C. Fisher. 1987. *Advancing the Study of Programming with Computer-aided Protocol Analysis*. Proceedings of the 2nd Workshop on Empirical Studies of Programmers, 1987, Ablex, Norwood, NJ, pp. 198–216.

[18] L. Cleveland. 1989. A program understanding support environment. *IBM Systems Journal*, 28(2), 324–344.

[19] M. T. Harandi and J. Q. Ning. 1990. Knowledge-based program analysis. *IEEE Software*, January, pp. 74–81.

[20] D. Hendrix, J. H. Cross II, and S. Maghsoodloo. 2002. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Transctions on Software Engineering*, 28(5), 463–477.

[21] G. W. Furnas. 1981. The fisheye view: A new look at structured files. In: *Readings in Information Visualization: Using Vision to Think* (Eds S. K. Card, J. D. Mackinlay, and B. Shneiderman), pp. 312–330, Morgan-Kaufmann.

[22] M. R. Jakobsen and K. Hornbaek. 2006. *Evaluating a Fisheye View of Source Code*. Proceedings of ACM CHI 2006 Conference on Human Factors in Computing Systems, 2006, ACM, New York.

[23] D. Sun and K. Wong. 2005. *On Evaluating the Layout of UML Class Diagrams for Program Understanding*. Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05), 2005, IEEE Computer Society Press, Los Alamitos, CA.

[24] M. Petre, A. Blackwell, and T. Green. 1998. Cognitive questions in software visualization. In: *Software Visualization: Programming as a Multimedia Experience* (Eds J. Stasko, J. Domingue, M. Brown, and B. Price), pp. 453–480. MIT Press.

[25] R. Wettel and M. Lanza. 2007. *Program Comprehension Through Software Habitability*. Proceedings of the 15th International Conference on Program Comprehension (ICPC'07), 2007, IEEE Computer Society Press, Los Alamitos, CA.

[26] R. P. Gabriel. 1996. *Patterns of Software*. Oxford University Press, New York, NY.

[27] T. J. Biggerstaff, B. Mitbander, and D. Webster. 1993. *The Concept Assignment Problem in Program Understanding*. Proceedings of the 15th International Conference on Software Engineering, 1993, IEEE Computer Society Press, Los Alamitos, CA.

[28] T. J. Biggerstaff, B. Mitbander, and D. Webster. 1994. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5), 72–83.

[29] N. Gold and K. Bennett. 2002. Hypothesis-based concept assignment in software maintenance. *IEE Proceedings-Software*, 149(4), 103–110.

[30] P. Tonella. 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6), 495–509.

[31] M. Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4), 352–357.

EXERCISES

1. What is meant by program comprehension?

2. Explain the two broad kinds of *knowledge* that programmers need to possess for program comprehension.

3. Explain the concept of *mental model* in program comprehension.

4. Identify and summarize the *static* elements of mental models.

5. Identify and summarize the *dynamic* elements of mental models.

6. Identify and summarize three *characteristics* of expert programmers from the standpoint of program comprehension.

7. Briefly explain the Letovsky model for program comprehension.

8. Briefly explain the Shneiderman and Mayer model for program comprehension.

9. Briefly explain the Brooks model for program comprehension.

10. Briefly explain the Soloway, Adelson, and Ehrlich model for program comprehension.

11. Briefly explain the Pennington model for program comprehension.

12. Briefly explain the Integrated metamodel for program comprehension.