Software Evolution and Maintenance

A Practitioner's Approach

Chapter 8

Program Comprehension

Outline of the Chapter

8.1 General Idea

8.2 Basic Terms

8.3 Cognition Models for Program Understanding

8.4 Protocol Analysis

8.5 Visualization for Comprehension

8.6 Summary

8.1 General Idea

- It is important to comprehend a complex system to be able to maintain it.
- Modification of software with inaccurate and incomplete understanding is likely to degrade its performance and reliability.
- Good program comprehension is key to providing effective software maintenance and effective evolution of software.

8.1 General Idea

- To understand the role of program comprehension, consider five kinds of tasks associated with program maintenance (Table 8.1.)

- Understanding the system or problem is common to all maintenance and evolution tasks.

- Understanding of a system is a cognitive issue and a number of cognitive models have been developed (Table 8.1.)

8.2 Basic Terms

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 Understand	Letovsky Brooks Shneiderman and Mayer

Table 8.2: Code cognition models [1] (©[1995] IEEE).

8.2 Basic Terms

- To understand the cognition models of Table 8.2, the following set of terms form the background material:
  - Goal of code cognition
  - Knowledge
  - Mental model

8.1 General Idea

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 coverage	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

Table 8.1: Tasks and activities requiring code understanding [1] (c [1995] IEEE).

8.2.1 Goal of code cognition

- A code maintainer tries to understand a program with a specific goal in mind.
  - Example 1: Debugging a program to detect the cause of a known failure
  - Example 2: Adding a new function to the existing program
- Identifying the goal can help in defining the scope of program comprehension.
- Scope of program comprehension: complete program or part of a program
- A program comprehension process is a sequence of activities that use existing knowledge about the program to generate new knowledge about it.
- Thus, program comprehension is a process of knowledge acquisition.

8.2.2 Knowledge

- Programmers possess two kinds of knowledge
  - General knowledge
  - Software-specific knowledge
- General knowledge: This covers a broad range of topics in computer systems and software.
  - Algorithms and data structures
  - Operating systems
  - Programming principles
  - Programming languages
  - Software architecture and design
  - Testing and debugging techniques

8.2.2 Knowledge

- Software-specific knowledge: This represents a detailed understanding of the software to be modified.
- Some examples of software-specific knowledge are
  - The software system has implemented public-key cryptography for data encryption.
  - The software system has been structured as a three-tier client-server system.
  - 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.

8.2.2 Knowledge

- In the process of gaining new knowledge about a software system, a programmer learns the details of the following aspects:
  - Functionality
  - Software architecture
  - Control flow and data flow
  - Exception handling
  - Stable storage
  - Implementation details
- A programmer goes back and forth between acquiring general knowledge and software-specific knowledge, as illustrated in Fig. 8.1.

8.2.2 Knowledge

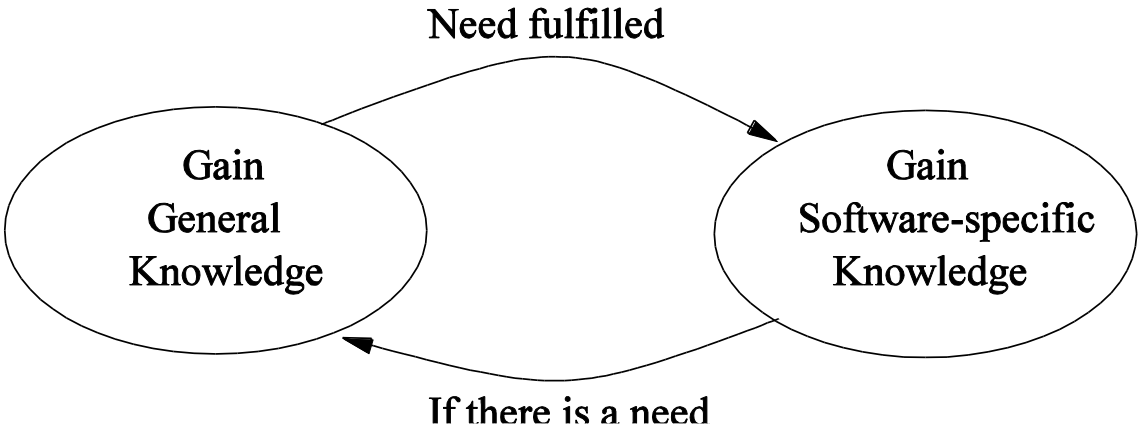

Figure 8.1: Gaining general knowledge and software-specific knowledge.

8.2.3 Mental model

- A mental model describes a programmer's mental representation of the program being comprehended.
- A mental model of a program is unique; different programmers view and interpret a programmer in different ways.
- A programmer develops a mental model by identifying both static and dynamic elements of the program.
- Examples of program elements
  - *for* loop
  - A TCP (Transmission Control Protocol) connection
  - Over all control flow

8.2.3 Mental model

- Static elements of a mental model
  - Text-structures
  - Chunks
  - Schemas
  - Plans
  - Hypotheses

- Dynamic elements of a mental model
  - Chunking
  - Cross-referencing
  - Strategies

8.2.3 Mental model (static elements)

- Text-structures
  - Code and its structure are known as text-structures.
  - It is useful in gaining control flow knowledge in program understanding.
  - A programmer can easily identify the following text-structures:
    - Loop constructs: *for*, *while*, and *until*
    - Sequences
    - Conditional statements: *if-then-else*
    - Variable definitions and initializations
    - Calling hierarchies within and among modules
    - Definitions of module parameters
  - Understanding text-structures is the beginning of program comprehension.

8.2.3 Mental model (static elements)

- Chunk
  - A program chunk is a block of related code segment.
  - Chunks enable programmers to create higher level abstractions from lower-level abstractions.
  - Examples
    - A code block initializing a module's parameters tells the programmer about the nature of the parameters and their value ranges.
    - Understanding a *for* loop enables a programmer to create an abstraction of an internal functional step performed by the program.
- Schema
  - Schemas are generic knowledge structures that guide the programmer's interpretations, inferences, expectations, and attentions when passages are comprehended.
  - The concept of programming plans correspond to the notion of schemas.

8.2.3 Mental model (static elements)

- Plans
  - A knowledge element is anything that is useful in understanding a program.
  - Plans are broad kinds of knowledge elements used by programmers.
  - Examples of knowledge elements
    - 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 block of comments describing a for loop
    - A *for* loop itself is a knowledge element.
    - A description of the problem domain of the program

8.2.3 Mental model (static elements)

- Plans
  - Example of plan
    - A doubly-linked list is an example of a plan; a designer has planned to implement certain concepts with this data structure.
  - A plan is a kind of schema with two parts:
    - Slot type
      - Slot types describe generic objects.
      - Example: A tree data structure is a generic slot type.
    - Slot filler
      - Slot fillers are customized to hold elements of particular types.
      - Example: A code segment, such as a *for* loop's code is a slot filler.
  - The programmer links the slot-type and slot-filler structures by means of the kind-of and is-a modeling relationships.

8.2.3 Mental model (static elements)

- Plans
  - There are two broad kinds of plans.
    - Domain plans
    - Programming plans
  - Domain plans
    - These include knowledge about the real world problem, including the program's environment.
    - Example: If the software is for numerical analysis application, plans will include schemas for different aspects of linear algebra, such as matrix multiplication and matrix inversion.
    - Domain plans help programmers understand the code.

8.2.3 Mental model (static elements)

- Plans
  - Programming plans
    - Programming plans are program fragments representing action sequences that programmers repeatedly apply while coding.
    - Example: 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.

8.2.3 Mental model (static elements)

- Hypotheses
  - As programmers start reading code and the related documents, they start developing an understanding of the program to varying degrees.
  - Programmers can test the results of their understanding as conjectures (aka hypotheses.)
    - Why: *Why* conjectures hypothesize the purpose of a program element.
      - Verification of a why conjecture enables a programmer to have a good understanding of the program element.
    - How: *How* conjectures hypothesize the method for realizing a program goal.
      - Given a program goal, the programmer needs to know how that goal has been implemented.
    - What: *What* conjectures enable programmers to classify program elements.
  - A conjecture may not be completely correct.
  - By continuously formulating and verifying conjectures, the programmer understands more and more code.

8.2.3 Mental model (dynamic elements)

- Dynamic elements
  - Chunking
  - Cross-referencing
  - Strategies

8.2.3 Mental model (dynamic elements)

- Chunking
  - In a program, the lowest level of chunks are code segments.
  - 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.
  - The process of chunking is 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.
  - A block of lower level labels can be replaced with one higher level label representing a higher level functionality.

8.2.3 Mental model (dynamic elements)

- 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.
  - Example:
    - Control flow and data-flow can be program elements at a lower level, whereas functionalities are higher level program elements.
    - There is a need to cross-reference between control-flow and data-flow elements and program functionalities.

8.2.3 Mental model (dynamic elements)

- Strategies
  - A strategy is a planned sequence of actions to reach a specific goal.
  - A strategy is formulated by identifying actions to achieve a goal.
  - 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 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.
    - Create a higher-level model of the function in terms of the chunks.
  - Strategies guide the two dynamic elements, namely, chunking and cross-referencing, to produce higher-level abstraction structures.

8.2.4 Understanding Code

- Two key factors influencing code understanding are:
  - Acquiring knowledge from code
    - Code is a rich source of information.
  - The level of expertise of the code reader
    - The level of expertise determines how quickly the code is understood.

8.2.4 Understanding Code

- Acquiring knowledge from code
  - Several concepts can be applied while reading code in order to gain a high-level understanding of programs.
    - Beacons
      - A beacon is code text that gives a cue to the computation being performed in a code block.
        - Example swap(), sort(), select(), startTimer().
      - Code with good quality beacons are easier to understand.
    - Rules of programming discourse
      - Rules of programming discourse specify the conventions, also called "rules," that programmers follow while writing code.
      - Some examples of rules are:
        - Function name: The function name agrees with what the function does.
        - Variable name: Choose meaningful names for variables and constants.
      - The rules set up expectations in the minds of a reader about what should be in the program.

8.2.4 Understanding Code

- Levels of expertise of code readers
  - Expert programmers tend to possess the following characteristics:
    - Organization of knowledge by functional characteristics
      - Novice programmers tend to organize program knowledge in terms of program syntax.
      - Experts tend to organize knowledge in terms of algorithms and functionalities.
    - Comprehension with flexibility
      - Experts tend to generate a breadth-first view of the program, and keep adding useful details as more information is available.
    - Development of specialized design schemas
      - Design schemas are used to organize complex entities into constituents.

8.3 Cognition Models for Program Understanding

- 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

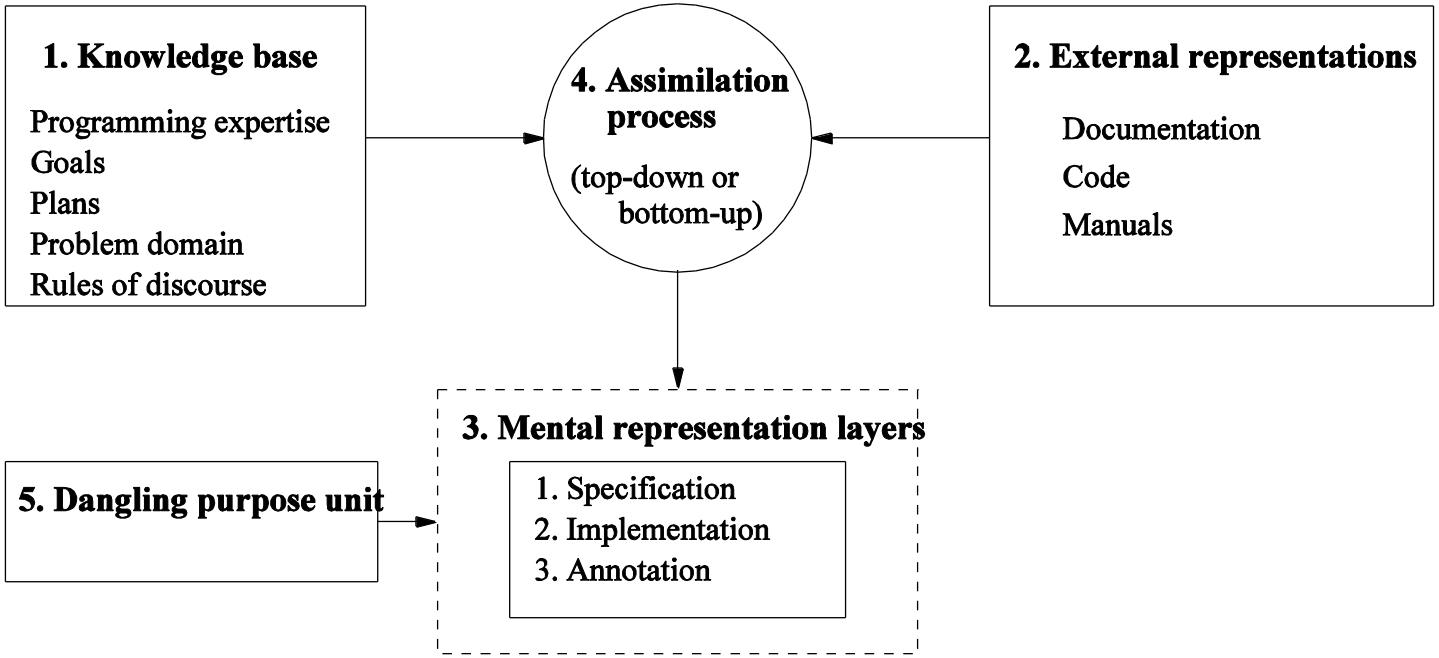

Figure 8.2: Letovsky's program comprehension model.

8.3.1 Letovsky model

1. Knowledge base
   - Programming expertise
     - Programming expertise helps in asking questions, making conjectures, and searching for specific information in the code.
     - The questions are grouped into five categories:
       - *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 as follows:
       - Why is the variable being reset to zero?
       - What is done to the memory block after the data is transmitted?
       - Why is the memory block being deleted in two places?

8.3.1 Letovsky model

- 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, to be able to create a higher level of abstraction.
- Problem domain
  - 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.
- Plans
  - Programmers have their own ways (also called plans) of finding solutions to problems.
  - They use widely used solutions to some common problems.

8.3.1 Letovsky model

- 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.

2. External Representations
   - The external representations of a program include its source code, documentation in the form of some comments, and manuals.
   - The manuals are useful in understanding the high-level goals of the code, whereas the in-line comments are useful in understanding the low-level details.

8.3.1 Letovsky model

3. Mental Representation

- By reading code and documents, a programmer may create the followings:
  - Specification of the program
    - Here, a specification means a complete and unambiguous description of the goals of the program. This is done by identifying the user-level functions, attributes of the functions, and program constraints.
  - High-level implementation of the program
    - This means producing a complete and unambiguous description of the actions and data structures of the program.
  - Annotation of the program
    - Make a two-way association between the goals and the actions and data structures, by annotating the program as follows:
      - ➢ How each goal in the specification is accomplished and by which actions and data structures.
      - ➢ What goals use the services of a given action or a data structure.
    - In other words, establish a traceability matrix between program goals and actions and datastructures.

8.3.1 Letovsky model

4. Assimilation Process
   - Programmers combine their knowledge base and the external representations to create their mental models. This process is known as *assimilation*.
   - The assimilation process can work in three ways: top-down, bottom-up, and opportunistic.
     - Top-down: Begin with a goal, followed by possible implementations of the goal.
     - Bottom-up: Identify program plans from code, make annotations, and move up to the top.
     - Opportunistic: Combine both top-down and bottom-up in an opportunistic manner.
5. Purpose Unit
   - This unit captures those goals whose implementations have not been clearly understood.

8.3.2 Shneiderman and Mayer model

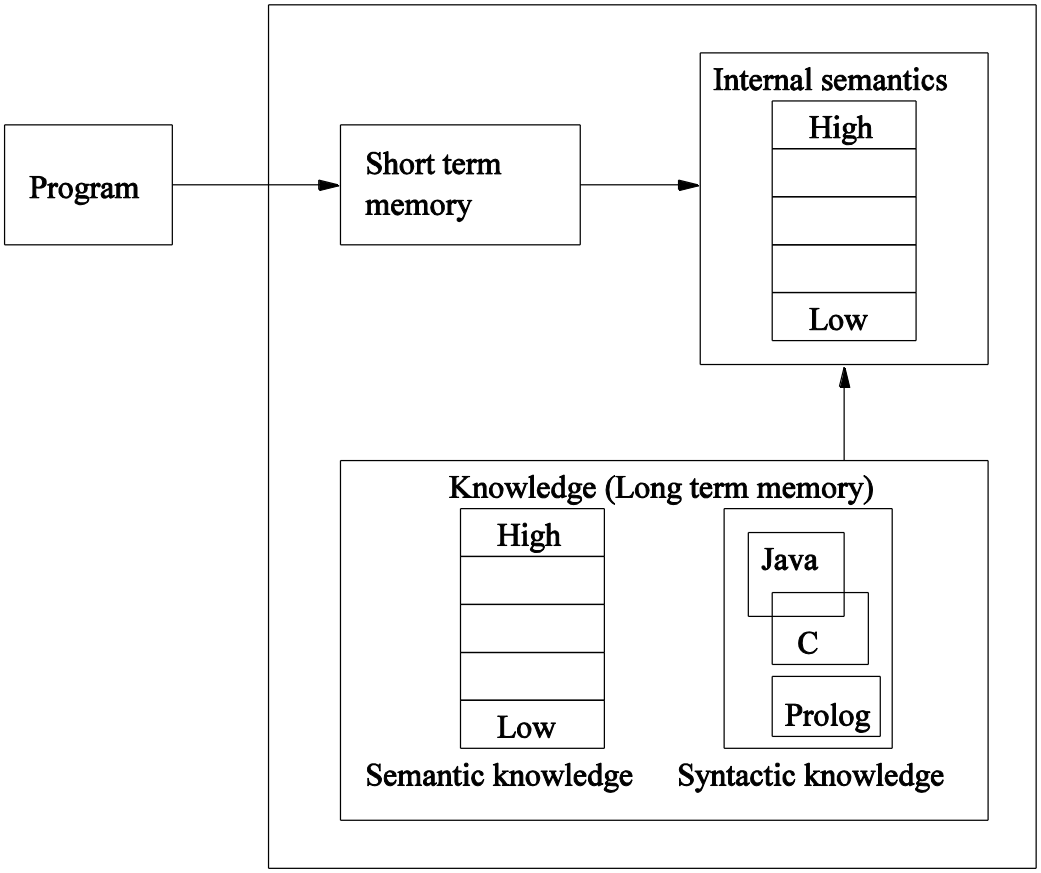

Figure 8.3: Shneiderman and Mayer program comprehension model.

8.3.2 Shneiderman and Mayer model

- The model comprises three key components
  - Short term memory of the programmer
  - The programmer's knowledge to understand the code
  - Internal semantics of the code as understood by the programmer
- Internal semantics
  - An internal semantics lies between the top-level goals of the program (aka the *what* aspects) and their detailed implementations.
  - In between the two extremes, programmers develop an internal semantic structure to represent the program.
  - An example of intermediate level abstraction is the concept of *call graphs*.

8.3.2 Shneiderman and Mayer model

- Knowledge
  - Here knowledge refers to the application domain knowledge and programming knowledge (both syntactic knowledge and semantic knowledge), which are stored in the long term memory of the programmer.
    - Syntactic knowledge concern individual programming languages.
    - Semantic knowledge means programming concepts and techniques at different levels of abstractions.
- Short term memory
  - 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.

8.3.3 Brooks comprehension model

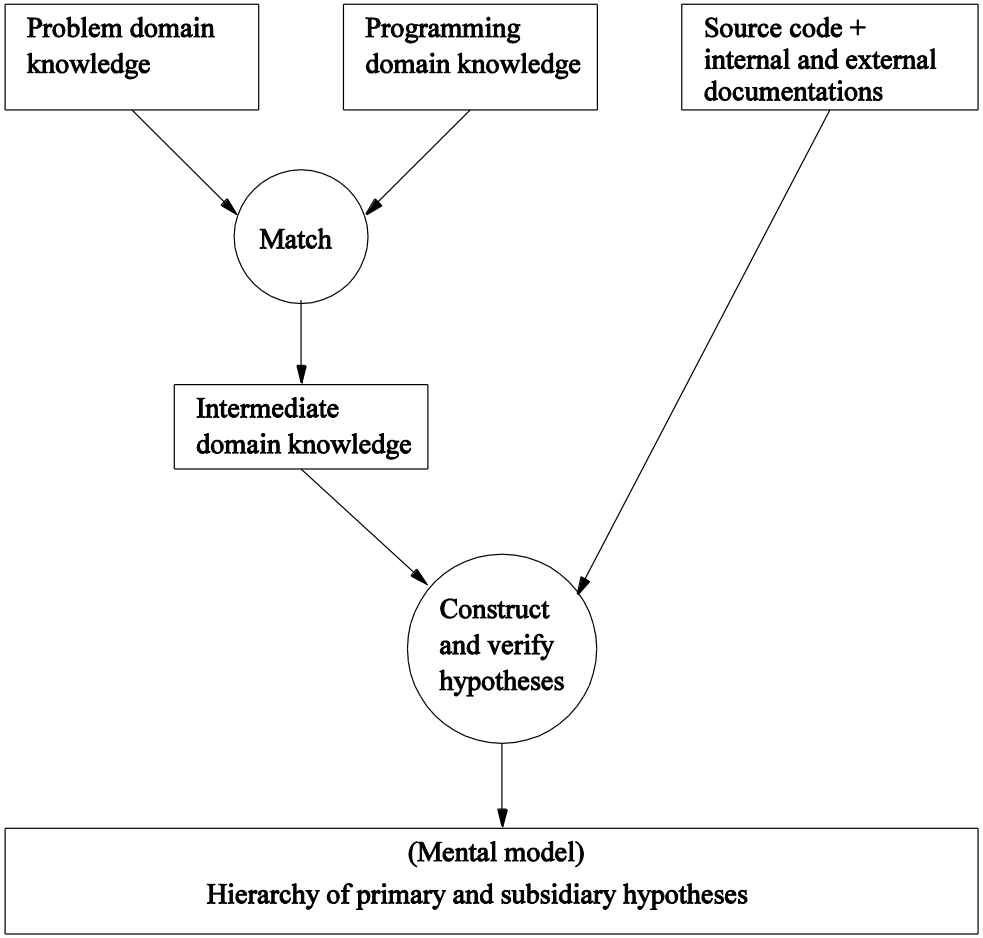

Figure 8.4: An overview of Brooks comprehension model.

8.3.3 Brooks comprehension model

- The three key elements of the model are
  - Code viewed as performing mappings from a problem domain to the programming domain
  - Understanding the mappings in terms of hypotheses
  - Verification and refinement of hypotheses

8.3.3 Brooks comprehension model

- The three key elements of the model are
  - 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.
    - 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.
    - 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.

8.3.3 Brooks comprehension model

- Understanding the mappings in terms of hypotheses
  - Programmers read all available documentations 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.
  - 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.
  - Hypotheses can be organized in a hierarchical manner to represent both the breadth and depth of comprehending the program.

8.3.3 Brooks comprehension model

- 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.
  - 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.
  - 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.

8.3.3 Brooks comprehension model

- Verification and refinement of hypotheses
  - 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.

8.3.3 Brooks comprehension model

- Brooks has identified a number of factors having an impact on program comprehension:
  - Characteristics of source code
  - Quality of documentation
  - Task differences affect comprehension
  - Programmers differ in their ability to comprehend programs

8.3.4 Soloway, Adelson, and Ehrlich model

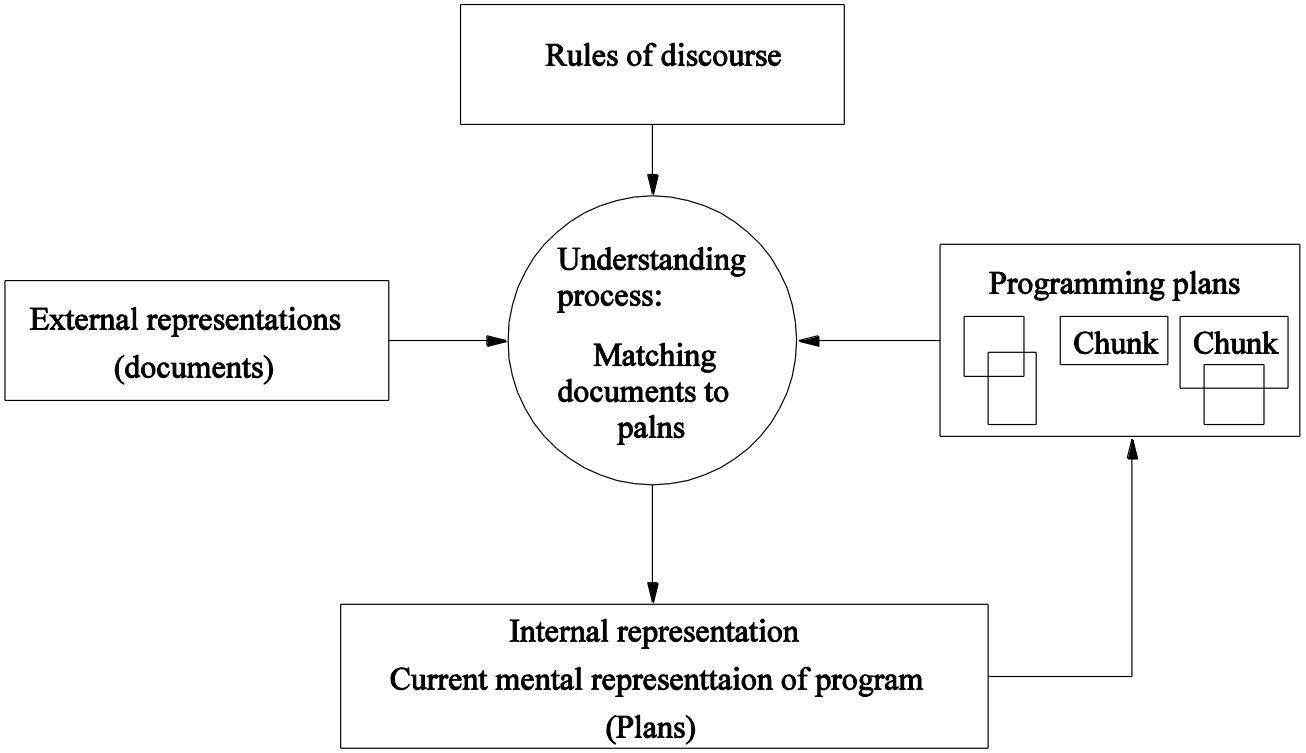

Figure 8.5: Soloway, Adelson, and Ehrlich comprehension model.

8.3.4 Soloway, Adelson, and Ehrlich model

- This model works in a top-down manner, and it applies when the code is familiar to the programmer.
- Two fundamental concepts in the model are
  - programming plans (aka schemas) and
  - programming rules of discourse.
- Programmers have and use specific programming plans and rules of programming discourse to comprehend programs.
- Some concrete rules of programming discourse are:
  - 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.

8.3.4 Soloway, Adelson, and Ehrlich model

- Similar to the other models, documentations play a key role in program understanding in this model.
- 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.
  - Comprehension begins with a high level program goal, and finer, lower level subgoals are generated to realize the upper level goals.
- Comprehension is an iterative process; 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 complete when the programmer has associated all the programming plans with the goal hierarchy.

8.3.5 Pennington Model

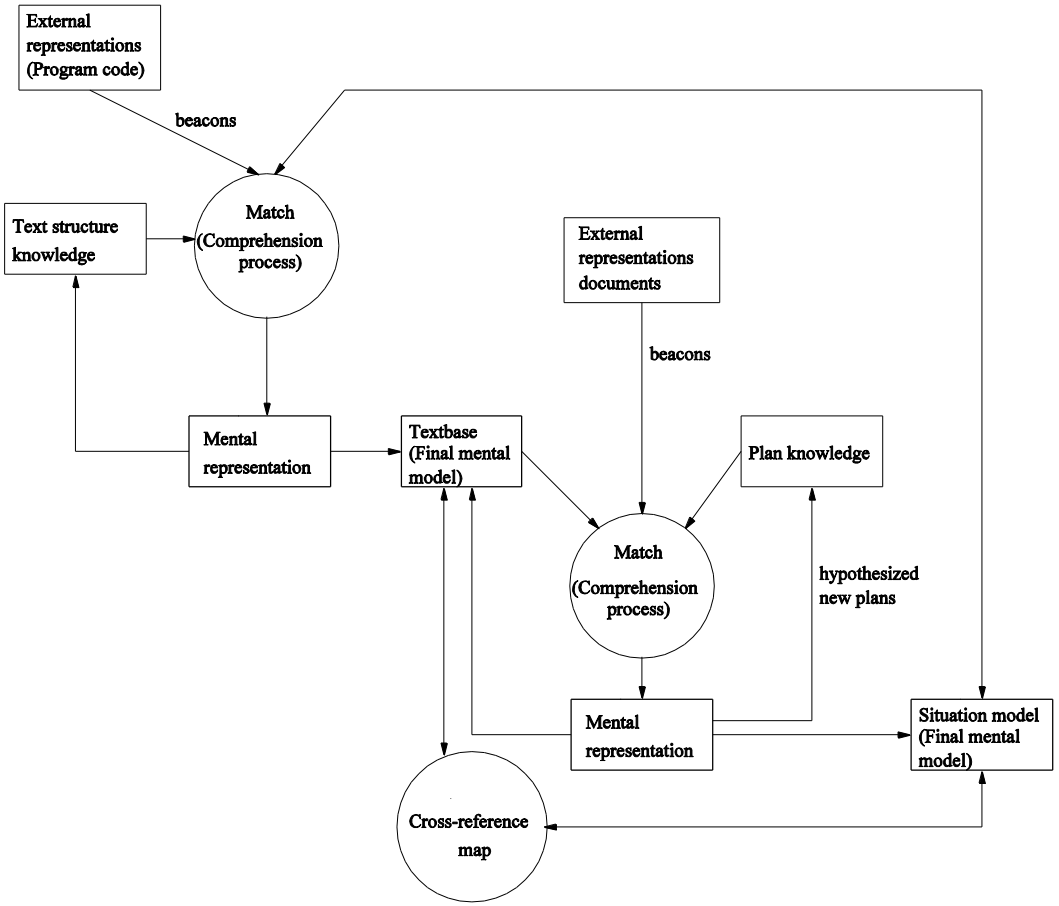

Figure 8.6: Pennington model.

8.3.5 Pennington Model

- The model applies two concepts:
  - textbase and
  - situation model. 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 representation box and the situation model.

8.3.5 Pennington Model

- The concepts of textbase and situation model are explained as follows:
  - Textbase (Program model)
    - 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.
    - 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.
  - Situation model
    - A situation model represents what the text is about.
    - The model requires knowledge of the real-world domains and objects.
    - Situation models are built via cross-referencing and chunking.

8.3.5 Pennington Model

- 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 documentations, plan knowledge, and the textbase to create the situation model.
  - The textbase and the situation model are cross-referenced to refine and update the two models.
- While reading code, programmers gain knowledge about the following aspects of code:
  - Operations
  - Control Flow
  - Data Flow
  - State
  - Function

8.3.6 Integrated Metamodel

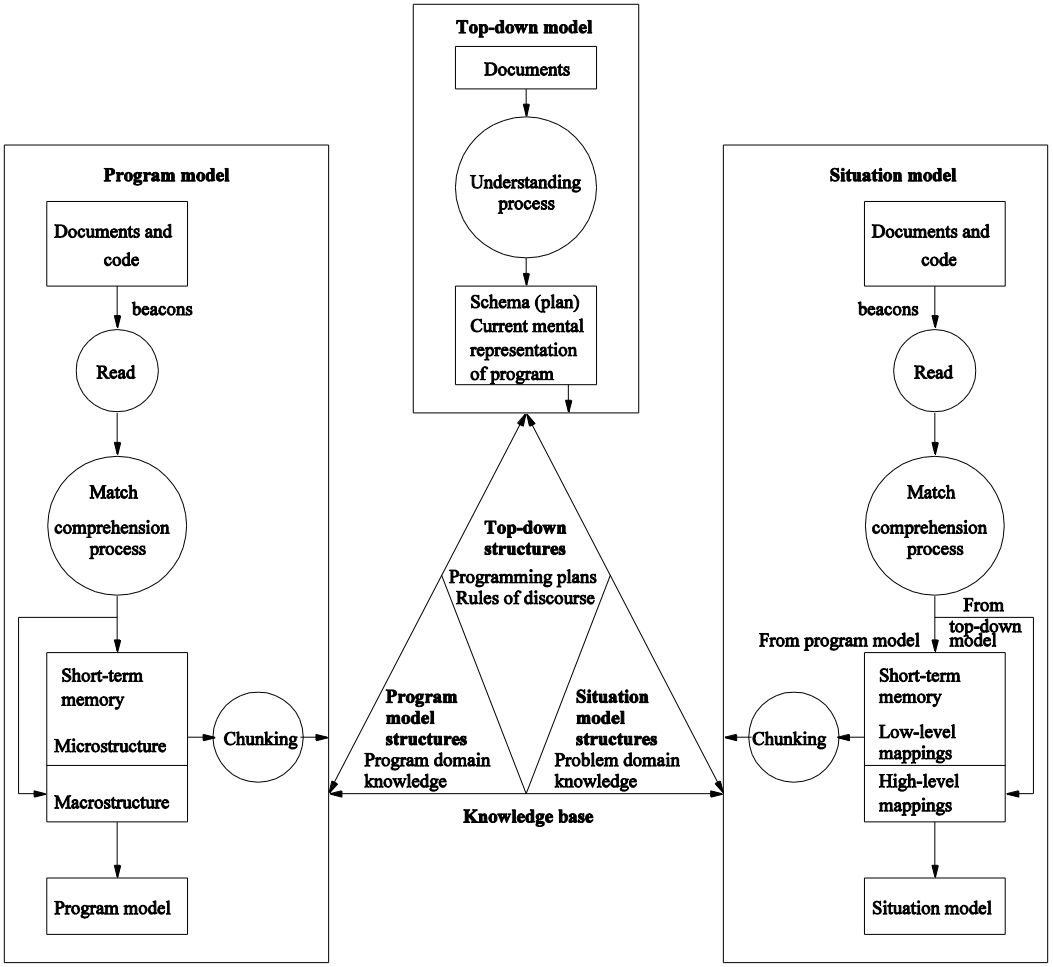

Figure 8.7: Integrated Metamodel [1] (c [1995] IEEE).

8.3.6 Integrated Metamodel

- The four major components of the model are explained in the following:
  - Top-down model
    - It is also known as *domain model* and it represents domain knowledge about the program.
  - Program model
    - If the programmer understands what the program is doing from the control flow aspect, then he has a program model of the code.
  - Situation model (See the Pennington's model)
    - After building a program model, a programmer builds a situation model by using control flow and data flow information in a bottom-up manner.
  - Knowledge base
    - The knowledge base provides a medium for interactions among the above three models, and comprises the following three kinds of knowledge structures. (See next slide.)

8.3.6 Integrated Metamodel

- Knowledge base (contd.)
  - The …. following three kinds of knowledge structures.
    - Top-down structures
      - These include programming plans and rules of discourse.
      - The programming plans are categorized into: strategic plans, tactical plans, and implementation plans.
    - Program model structures
      - These include two kinds of knowledge: text-structure knowledge and plan knowledge.
    - Situation model structures
      - These are described in terms of problem domain knowledge and functional knowledge.

8.3.6 Integrated Metamodel

- In the integrated metamodel, all the three models are simultaneously built.
- The programmer takes an opportunistic approach and simultaneously builds 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 (aka plans).

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:
  - 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?
- *Protocol analysis*, studied in the field of psychological research, is a key concept used in finding answers to the above questions.

8.4 Protocol Analysis

- *Protocol analysis* is a methodology for eliciting verbal reports from participants (programmers in this case) about their thought sequences as a valid source of data on thinking.
- Protocol analysis is composed of two steps:
  - Concurrent verbalization of a comprehension task: This step produces textual data (aka protocol data) representing the thought sequence of a programmer as he performs the comprehension task by reading the code.
  - Analysis of protocol data: The protocol data is analyzed to understand the characteristics of the thinking performed by the programmer.

8.4 Protocol Analysis

- Concurrent verbalization of a comprehension task
  - Programmers are asked to verbalize their thoughts while working on a specific task, and it is recorded on audio-visual systems.
    - Programmers are asked to "think aloud": they say loudly everything they think, evaluate, and (mentally) move.
    - This is called *think aloud protocol* (TAP)
  - Some examples of concurrent verbalization are:
    - 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*.
    - ….

8.4 Protocol Analysis

- 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. This is called *encoding*.
      - Coding categories are selected with a model of the verbalization process in mind.
      - Coding system of Ericsson and Simon: There are four kinds of segments: intentions, cognitions, planning, and evaluations.
    - 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

8.5 Visualization for Comprehension

- (Explained in Sec. 4.6.6 and applied in reverse engineering.)
- Visualization is supported with tools for program comprehension.
  - PUNS (Program Understanding Support environment)
  - PAT (Program Analysis Tool)
  - Fisheye view
  - UML (Unified Modeling Language)
  - City metaphor

8.5 Visualization for Comprehension

- PUNS (Program Understanding Support environment)
  - Developed at IBM to provide *multiple views* of a program.
    - A Call graph for a set of procedures
    - A Control flow graph for an individual procedure
    - A graph showing the relationship between a file and a procedure that uses it
    - A data flow graph
    - 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.

8.5 Visualization for Comprehension

- Program Analysis Tool (Harandi and Ning)
  - Presents a heuristic-based concept recognition mechanism to extract high-level functional concepts from source code.
  - Assists programmers answer the following questions:
    - What high-level concepts does the program implement?
    - How are the high-level concepts coded in terms of low-level details?
  - Explicitly represents two types of knowledge:
    - Program knowledge
      - Represented by programming concepts found in the code.
    - Analysis knowledge
      - Represented by information contained in program plans.
  - Manages two databases to manipulate the two types of knowledge:
    - A data base of coding heuristics, data structure definitions, and functional coding pattern.
    - A data base of rules for program plans covering value accumulation, counting, sequential search of ordered and unordered structures, different types of searching, and sorting.

8.5 Visualization for Comprehension

- Fisheye View (Jakobsen and Hornbaek)
  - It supports programmer's navigation and comprehension.
  - A fisheye view displays those parts of the source code that have the highest degree of interest related to the current focus of the programmer.
  - Shows both *overview* and *details*.
    - 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.
    - The portion of the code shown in the detail area is visually connected with its location in the overview.
  - The fisheye interface of Jakobsen and Hornbaek possess the following features:
    - Focus and context area
    - Degree of interface function
    - Magnification function
    - User interaction

8.5 Visualization for Comprehension

- UML
  - *Class diagrams* can aid programmers in code comprehension.
  - The concepts of *perceptual organization* and *perceptual segregation* can be applied to organize UML class diagrams.
    - Perceptual organization indicates when entities are organized in near proximity.
    - Perceptual segregation indicates when entities are separated.
  - The followings are some important principles of perceptual organization:
    - Good figure
    - Similarity
    - Proximity
    - Familiarity (Meaningfulness)
    - Element connectedness

8.5 Visualization for Comprehension

- UML
  - Perceptual segregation is further explained as follows:
    - When one looks at the environment, what is seen is a whole picture – and not separate parts.
    - The following factors make an entity more like a figure that can be easily recognized.
      - Symmetry
      - Orientation
      - Contours

8.5 Visualization for Comprehension

- City Metaphor (Wettel and Lanza)
  - This is a 3-dimensional visualization concept.
  - Classes are represented as buildings located in city districts which in turn represent packages.
  - The concept of habitability 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.
  - The concept of locality is supported by providing a navigable environment.

8.6 Summary

- Basic Terms
  - Knowledge
  - Mental model
    - Static elements: text-structures, code chunks, schemas, plans, andhypotheses
    - Dynamic elements: chunking, cross-referencing, and strategies
- Cognition Models for Program Understanding
  - Letovsky model, Shneiderman and Mayer model, Brooks model, Soloway, Adelson and Ehrlich model, Pennington model, and Integrated metamodel.
- Protocol Analysis
  - Think Aloud Protocol
- Visualization for Comprehension
  - PUNS, PAT, Fisheye view, UML, and City metaphor