



# Software Evolution : TOC

---

1. Introduction to Software Evolution
2. Taxonomy of Software Maintenance and Evolution
3. Evolution and Maintenance Models
4. Reuse and Domain Engineering
5. Program Comprehension
6. Impact Analysis
7. Refactoring
8. Reengineering
9. Legacy Information Systems

# Program Comprehension

---

- ❑ Modification of software with **inaccurate** and **incomplete understanding** is likely to **degrade** its performance and reliability
- ❑ The program comprehension **process** is one of the most important parts of software maintenance.
- ❑ A program comprehension process is a sequence of activities that **use existing knowledge** about the program to **generate new knowledge** about it.
- ❑ Program comprehension is estimated that **50 % to 90 %** of the time in software maintenance is consumed on **program comprehension** (Mateis et al. , 2000)

# Program Comprehension - Goal

---

- ❑ Program comprehension is a process of **knowledge acquisition**
- ❑ **Scope** of program comprehension: **complete** program or **part** of a program
- ❑ 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.

# Program Comprehension - Knowledge

---

- ❑ Two kinds of knowledge: **General** knowledge and **Software-specific** knowledge
- ❑ General knowledge covers a **broad range of topics** in computer systems and software, such as :
  - Algorithms and data structures
  - Programming principles and Programming languages
  - Software architecture and design
  - Testing and debugging techniques

# Program Comprehension - Knowledge

---

- ❑ Software-specific knowledge 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.

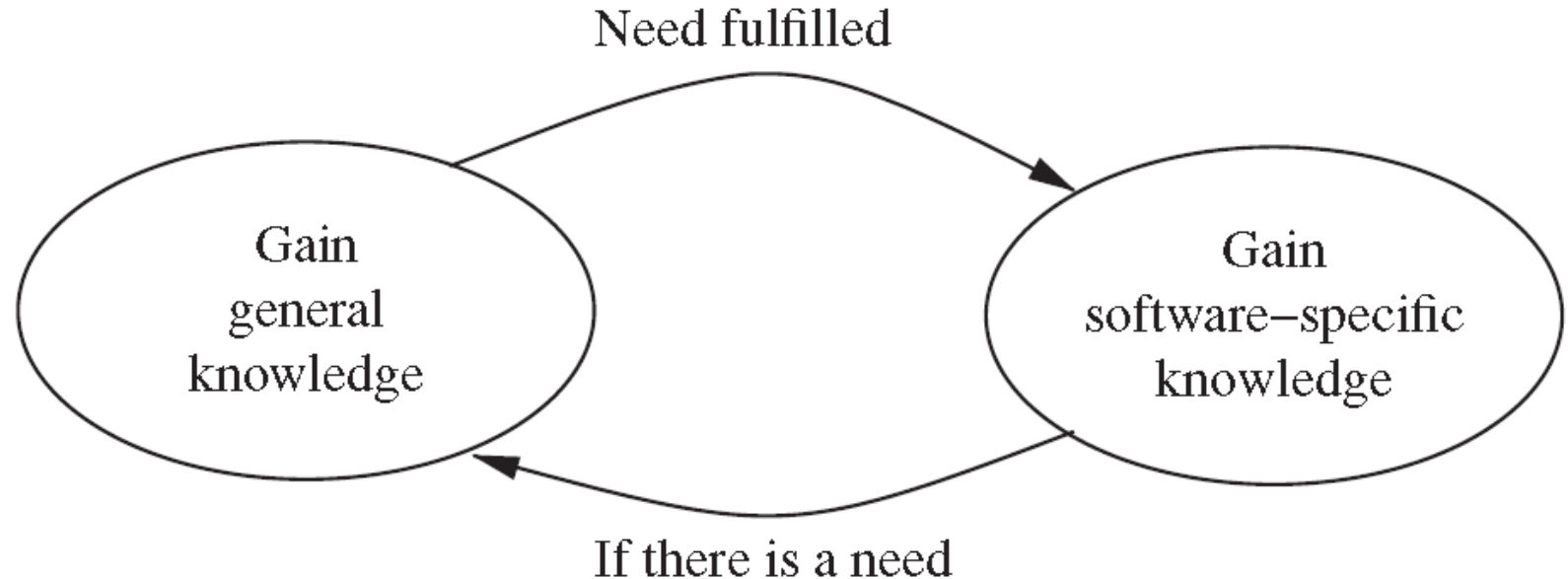
# Program Comprehension - Knowledge

---

❑ A programmer/maintainer goes back and forth between acquiring general knowledge and software-specific knowledge

❑ A programmer/maintainer learns the details of the following aspects:

- Functionality
- Software architecture
- Control flow and data flow
- Exception handling
- Stable storage
- Implementation details



# Program Comprehension - Mental Model

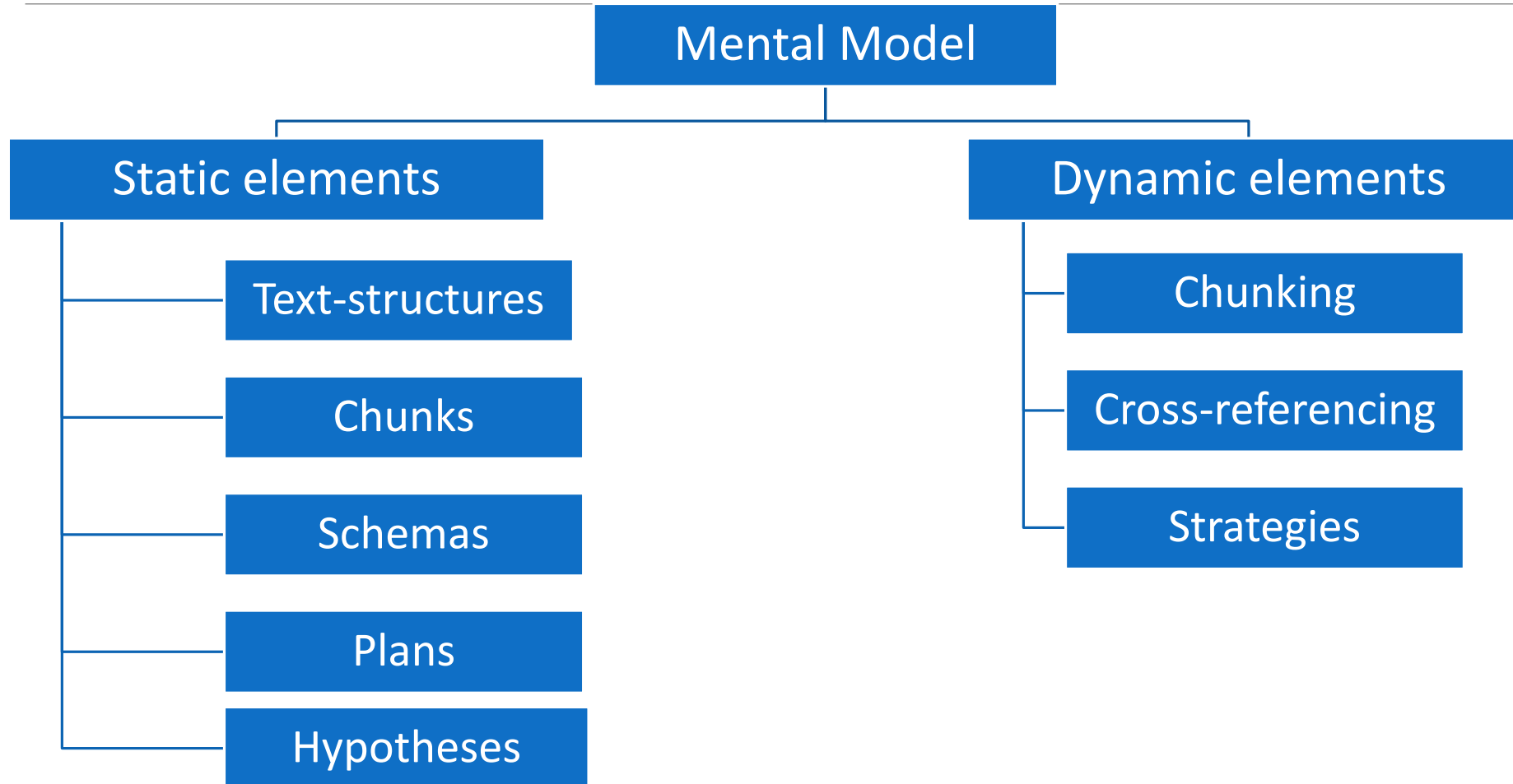
---

- ❑ A mental model **describes a programmer's/ maintainer's** mental representation of the program being comprehended.
- ❑ A mental model of a program is **not unique**; different programmers view and interpret a program 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
  - Cross-referencing



# Program Comprehension - Mental Model

---



# Mental Model – Text Structures

---

□ Text-structures denote code and its structure, It is useful in gaining control flow knowledge in program understanding.

□ The following text-structures can be identified :

- 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.

## Example: increment array items

```
int i = 0;
while (i <= a.length) {
    a[i] = a[i] + 1;
    i = i + 1;
}
```

# Mental Model - Chunk

---

- ❑ A program chunk is a block of related code segment.
- ❑ Chunks enable programmers to create higher level abstractions from lower-level abstractions.

A code block initializing tells  
the nature of the parameters  
and their value ranges

```
c = head();  
s = null;  
while (c <> null && s == null) {  
    if (c.value().matches("^*fred*"))  
        s = c;  
    c = c.next();  
}
```

Understanding the while  
loop enables to create an  
abstraction

# Mental Model - Schema

---

- ❑ Schemas are **generic knowledge** structures that guide the programmer's **interpretations, inferences, expectations, and attentions** when chunks are comprehended.

# Mental Model - Plan

---

- ❑ Plans are broad kinds of **knowledge elements**. A knowledge element is anything that is useful in understanding a program
- ❑ 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 and Slot filler
  1. Slot types describe generic objects.
    - ❑ Example: A **tree** data structure is a generic **slot type**.
  2. 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** modelling relationships.

# Mental Model - Plan

---

□ There are two broad kinds of plans: **Domain** plans and **Programming** plans

□ Domain plans

- These include **knowledge** about the **real world problem**, including the program's environment.
- Example: in a numerical analysis application, plans will include schemas for different aspects of **linear algebra**, such as **matrix multiplication** and **matrix inversion**.

# Mental Model - Plan

---

## □ Programming plans

- Programming plans are program fragments representing action sequences that repeatedly applied 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.

# Mental Model - Hypotheses

---

- Programmers can test the results of their understanding as hypotheses or conjectures
  - **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**.
- Use hypotheses to refine view of system incrementally



# Mental Model - Chunking

---

- ❑ The **process** of **creating** higher level chunks is called **chunking**. 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.
  - ❑ When a block of code is comprehended, 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.
  - ❑ The process **of chunking is iterative** to create increasingly higher levels of abstractions.

# Mental Model - Cross Referencing

---

- ❑ Cross-referencing means being able to **link elements** of different abstraction levels. This helps in building the **mental model** of the program under study.
- ❑ cross referencing is particularly helpful when **tracking down problems** in a large program
- ❑ Examples are:
  - Jump To Function, Variable Or Type Definition
  - List Uses Of Function, Variable Or Type
  - Determination Of Dependencies Between Separate Source Files

# Mental Model - Strategies

---

- ❑ A strategy is a **planned sequence** of **actions** to reach a specific goal.
- ❑ Example: if the goal is to understand the code representing a function, one can define a strategy as follows:
  1. Understand the **overall functionality** of the function by reading its specification (if it exists).
  2. Understand all the **input parameters** to the function.
  3. Read all **code** line by line.
  4. Identify **chunks** of related **code**.
  5. 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

# Acquiring knowledge From Code

---

❑ Several concepts can be applied while reading code in order to gain a high-level understanding of programs.

## 1. 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.

## 2. 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 eg. **openTCPconnection()**
  - ❑ **Variable name**: Choose meaningful names for variables and constants e.g. **MAX\_USERS**
  - ❑ **The rules set** up expectations in the minds of a reader about what should be in the program.

# Approaches to Code Comprehension

---

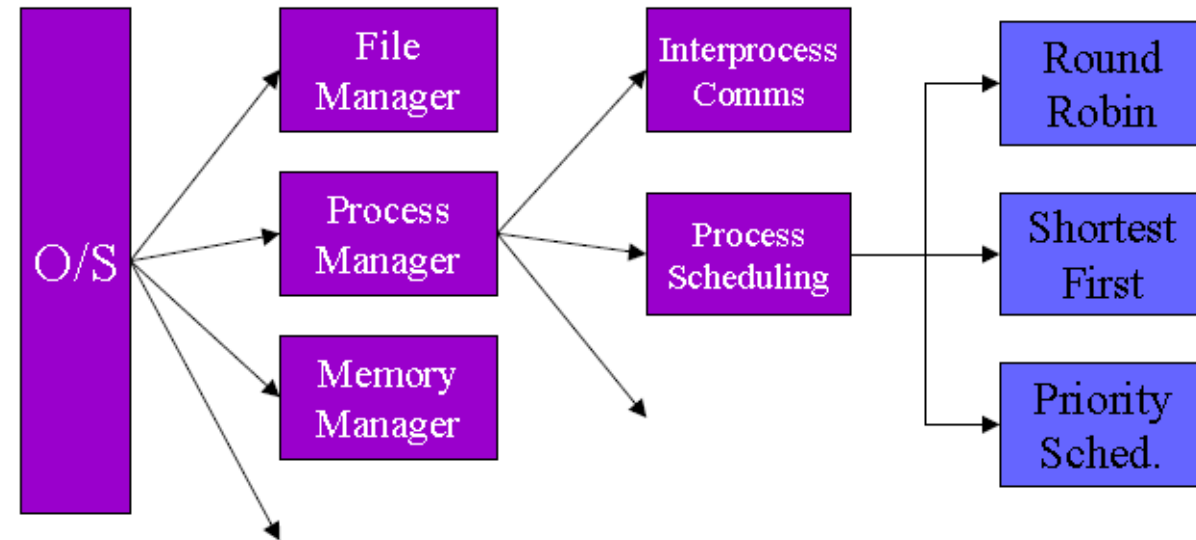
- ☐ Top- Down
- ☐ Bottom Up
- ☐ Opportunistic

# Approaches to Code Comprehension - Top Down Approach

❑ Suitable when programmer is familiar with the type of program to be comprehended

- E.g. may have worked on several similar systems
- Knows what components must be present
- Knows what these components consist of
- Use hypotheses to refine view of system incrementally

❑ Beacons are very important for this process



# Approaches to Code Comprehension - Bottom-Up Approach

---

□ Typically used when unfamiliar with the code/application

1. Look for **recognisable idioms** within the code

■ E.g. the “swap” idiom

- `t = x;`
- `x = y;`
- `y = t;`

2. **Combine recognised** units to understand even larger sections of the code

# Approaches to Code Comprehension - Bottom-Up Approach

A code block initializing tells the nature of the parameters and their value ranges

```
public String mystr(String original, char strchar) {  
    String returnstring = new String();  
    char p = ' ';  
    int a = 0;  
    int fl = 0;  
    while (original != null && a < original.length() && fl != 1) {  
        p = original.charAt(a);  
        if (p != strchar) {  
            returnstring = returnstring + p;  
        } else {  
            fl = 1;  
        }  
        a++;  
    }  
    return returnstring;  
}
```

Understanding *If* statement then the *while* loop enables to create an abstraction



# Approaches to Code Comprehension - Opportunistic Approach

---

□ Opportunistic Approach is a hybrid of the two

1. Begin with top-down, gain an overview of the functions of the program
2. Then selectively apply bottom-up strategies when nearing “code level”
  - to verify hypotheses resulting from top-down reading
3. Presence of beacons can indicate opportunity for change of strategy
  - E.g. by suggesting a hypothesis that is best verified by non-current strategy

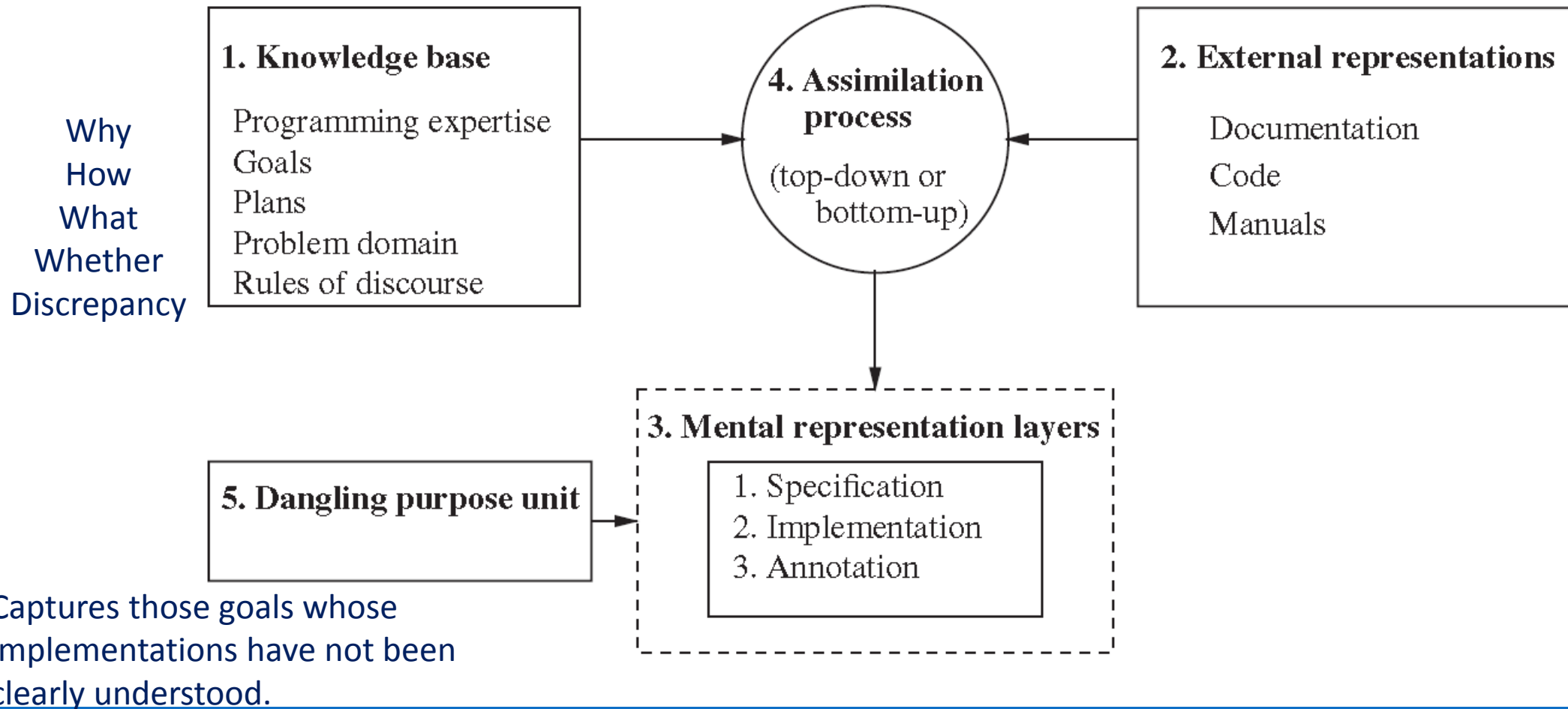
# Cognition Models For Program Comprehension

---

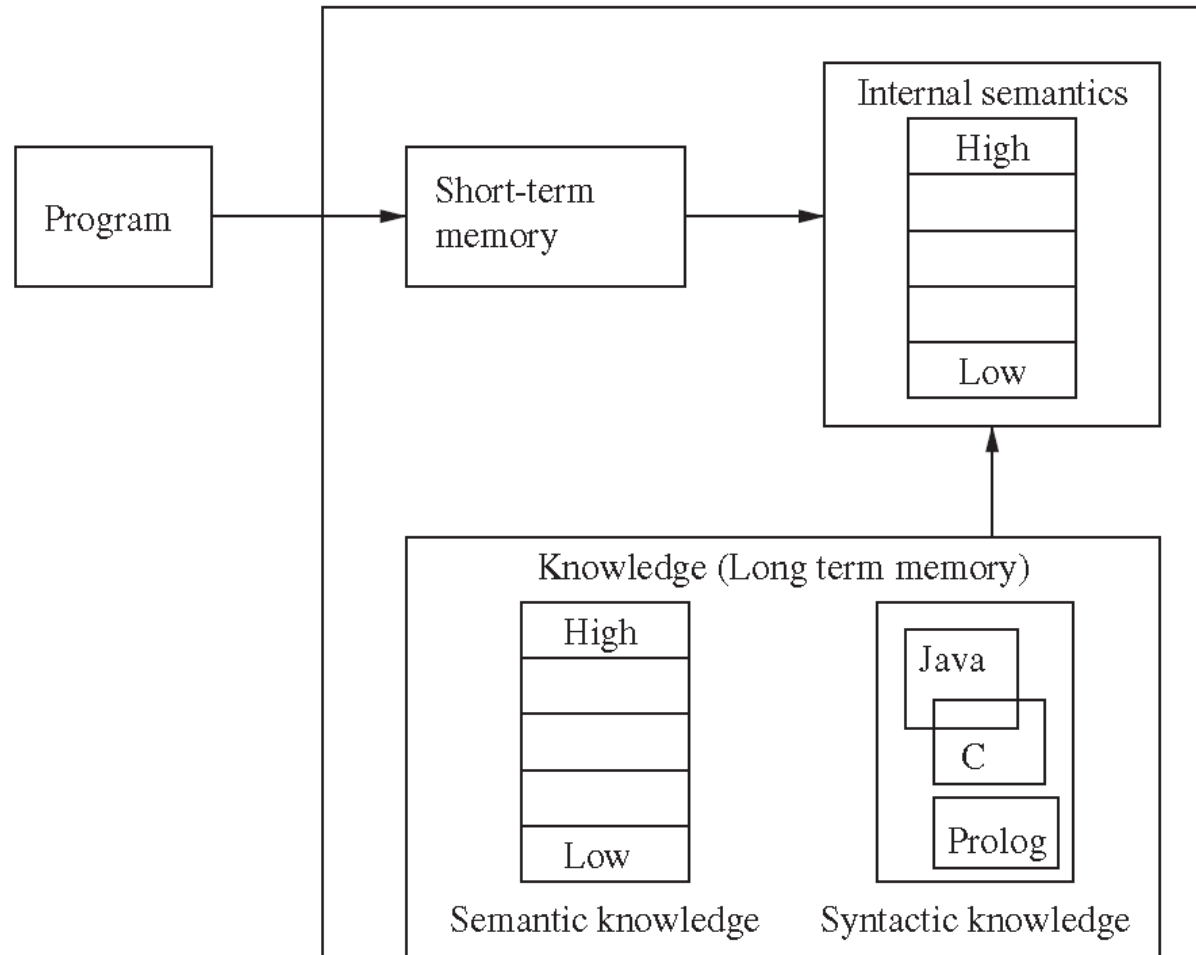
Several cognition models have been proposed the commonly referenced models are:

1. Letovsky model
2. Shneiderman and Mayer model
3. Brooks model
4. Soloway, Adelson, and Ehrlich model (top-down model)
5. Pennington model (bottom-up model)
6. Integrated metamodel

# Letovsky Code Comprehension Model



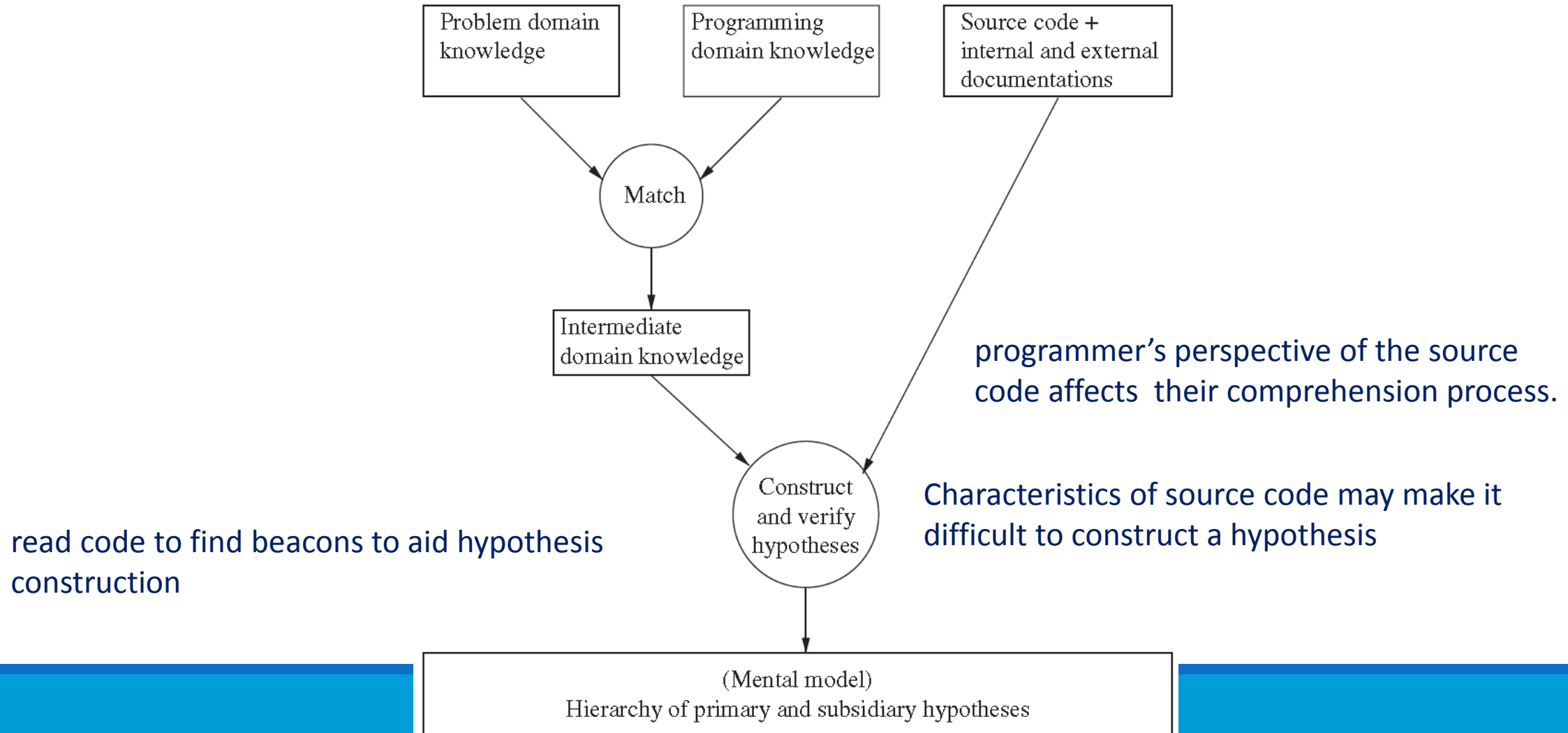
# Shneiderman and Mayer Comprehension Model



**Call graphs**—what modules are called by a given module and what modules use the services of the given module.

application domain knowledge and programming knowledge

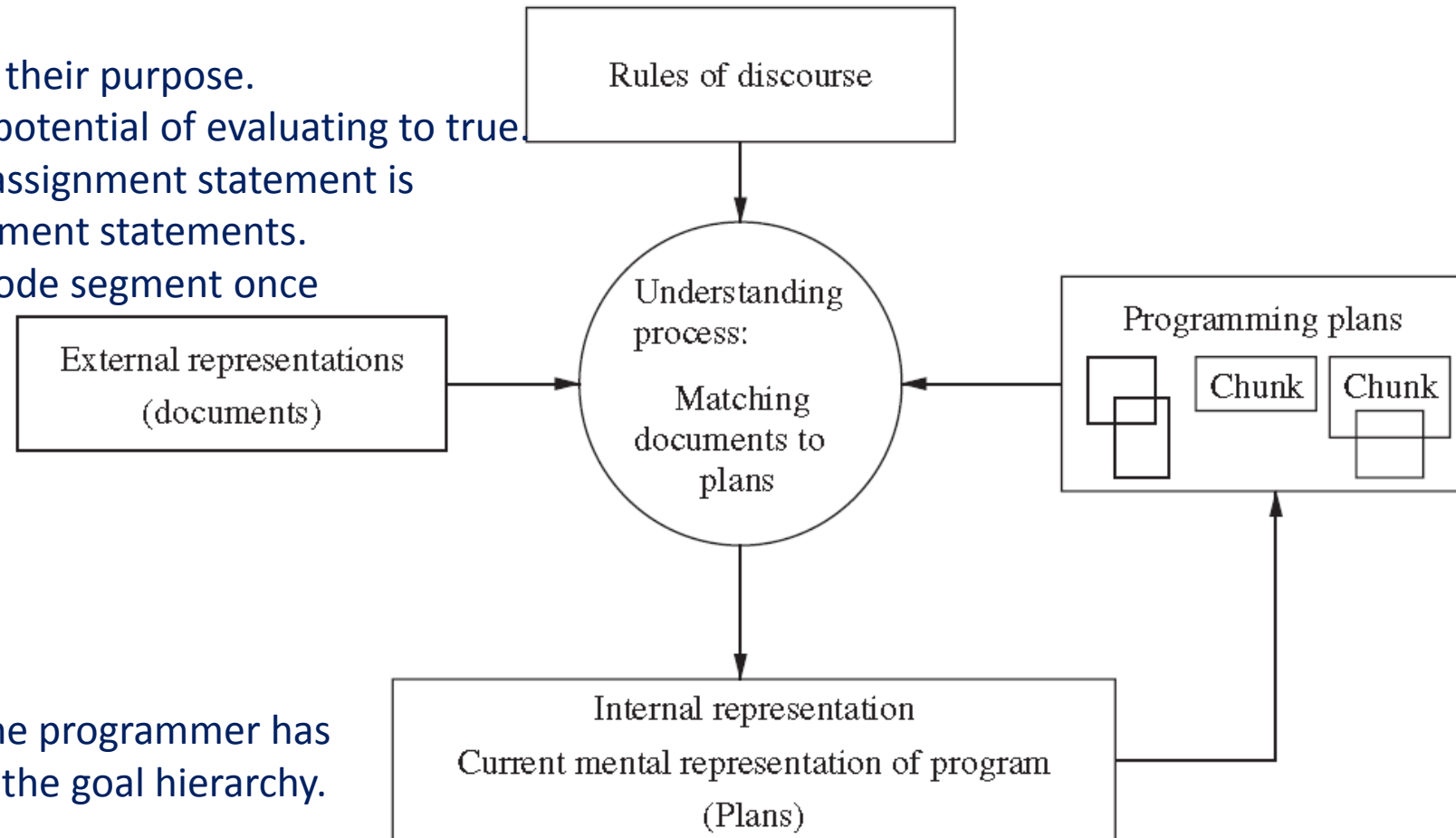
# Brooks Comprehension Model



# Soloway, Adelson, and Ehrlich Comprehension Model

## Example of rules of discourse

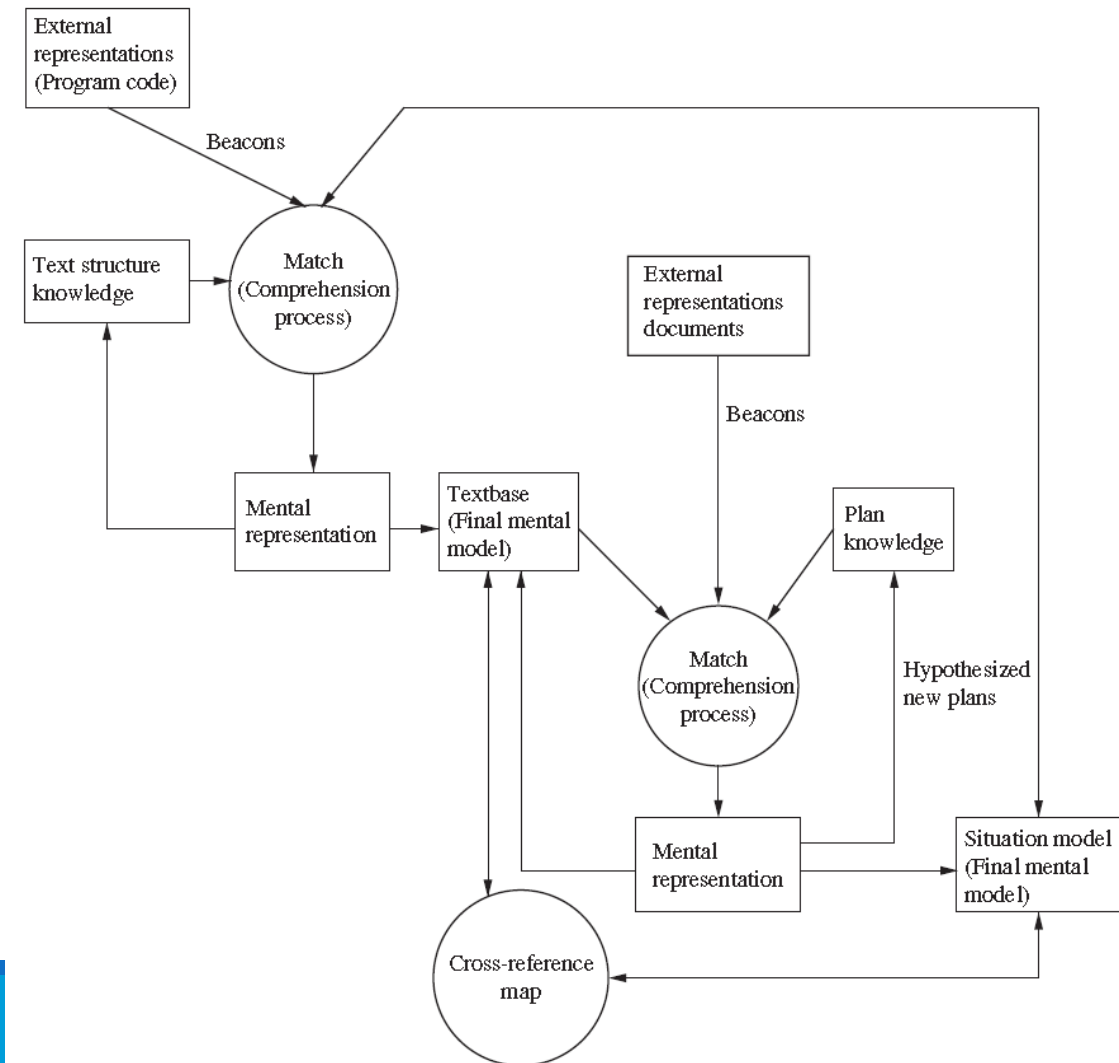
- The names of the variables reflect their purpose.
- A tested condition must have the potential of evaluating to true.
- A variable that is initialized by an assignment statement is subsequently updated with assignment statements.
- Use an if statement to execute a code segment once

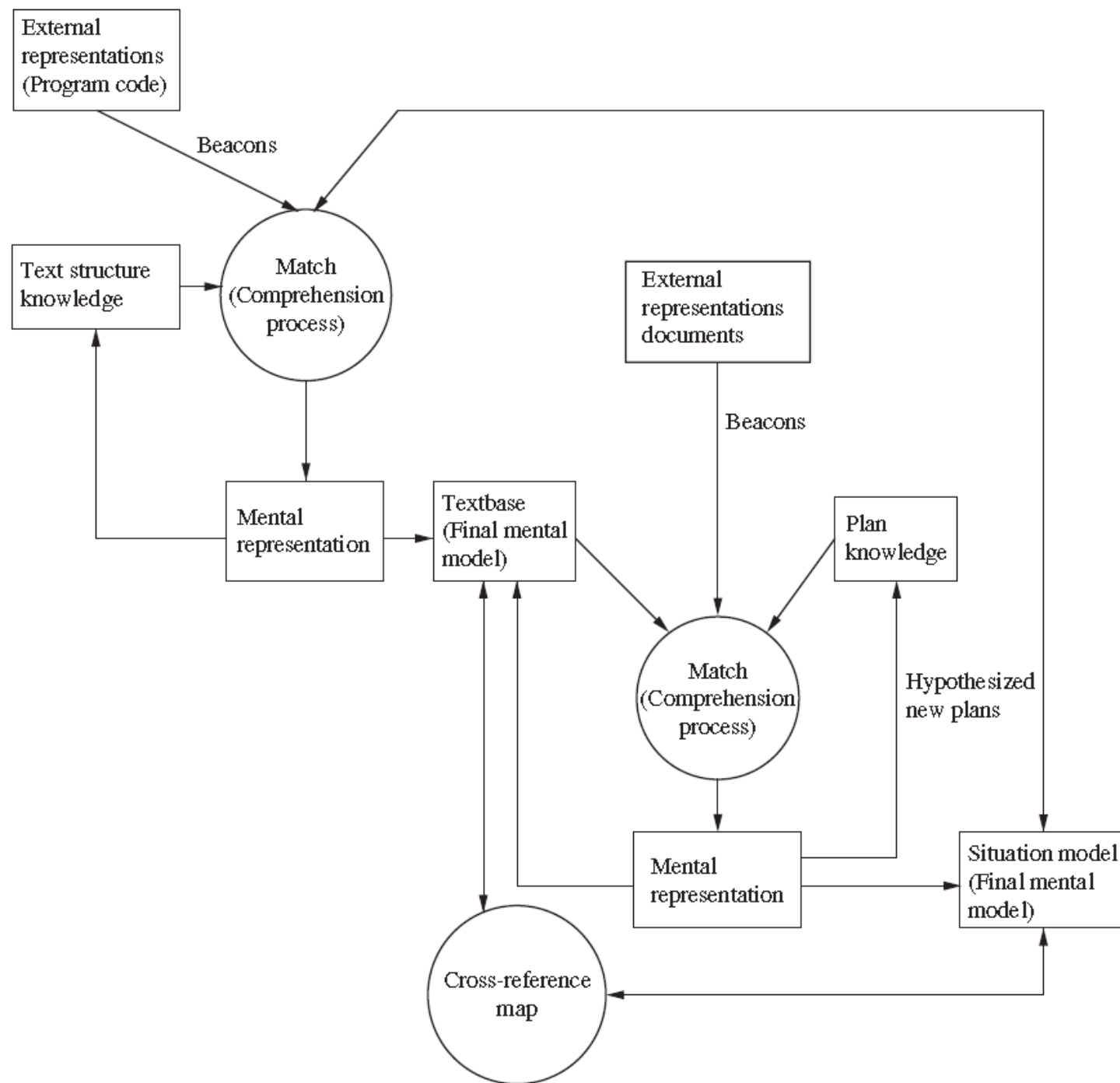


The process is said to be complete when the programmer has associated all the programming plans with the goal hierarchy.

# Pennington Comprehension Model

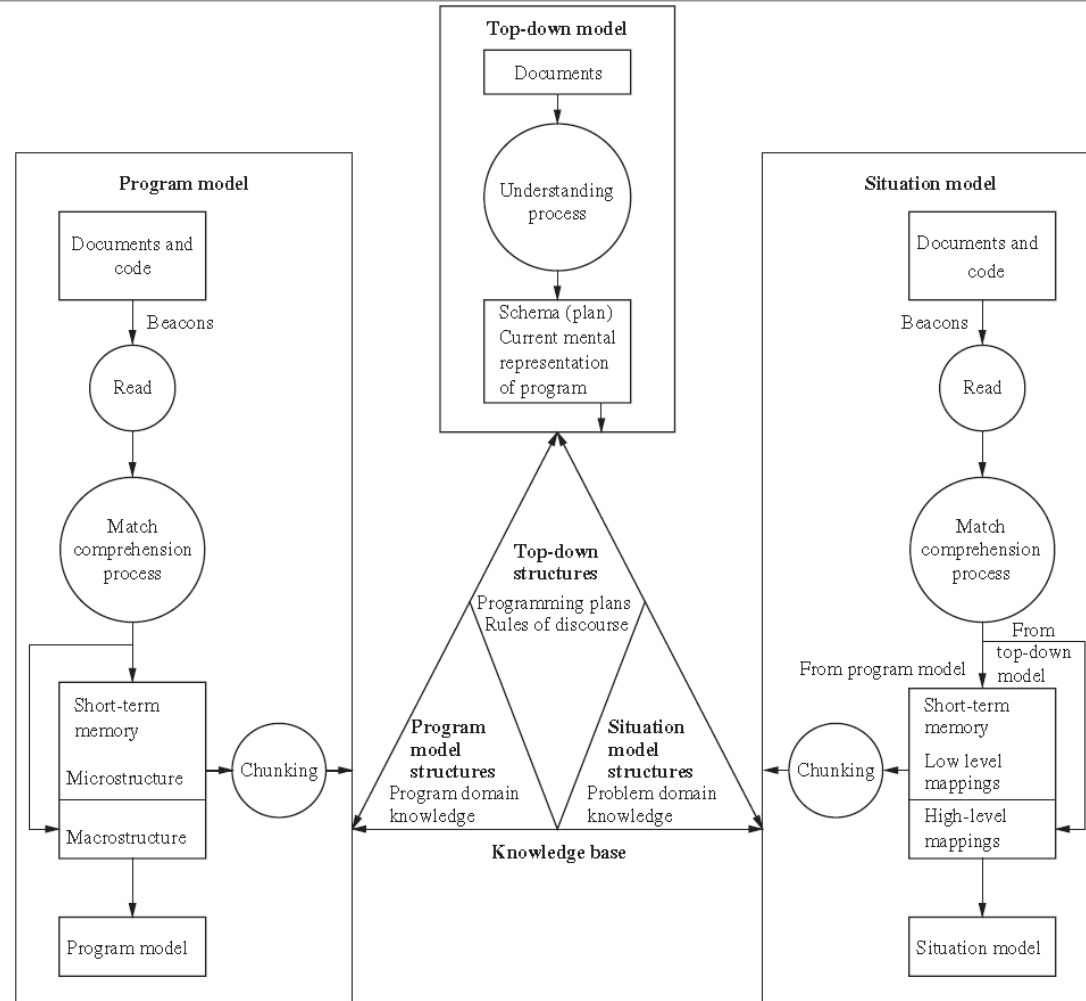
- The programmer assimilates their understanding of the code, knowledge of the text structure, and the situation model to create a **mental model** in the form of a **textbase**.
- The programmer assimilates the documentations, plan knowledge, and the textbase to create the **situation model**.
- The **textbase** and the **situation model** are **cross-referenced** to refine and update the two models.

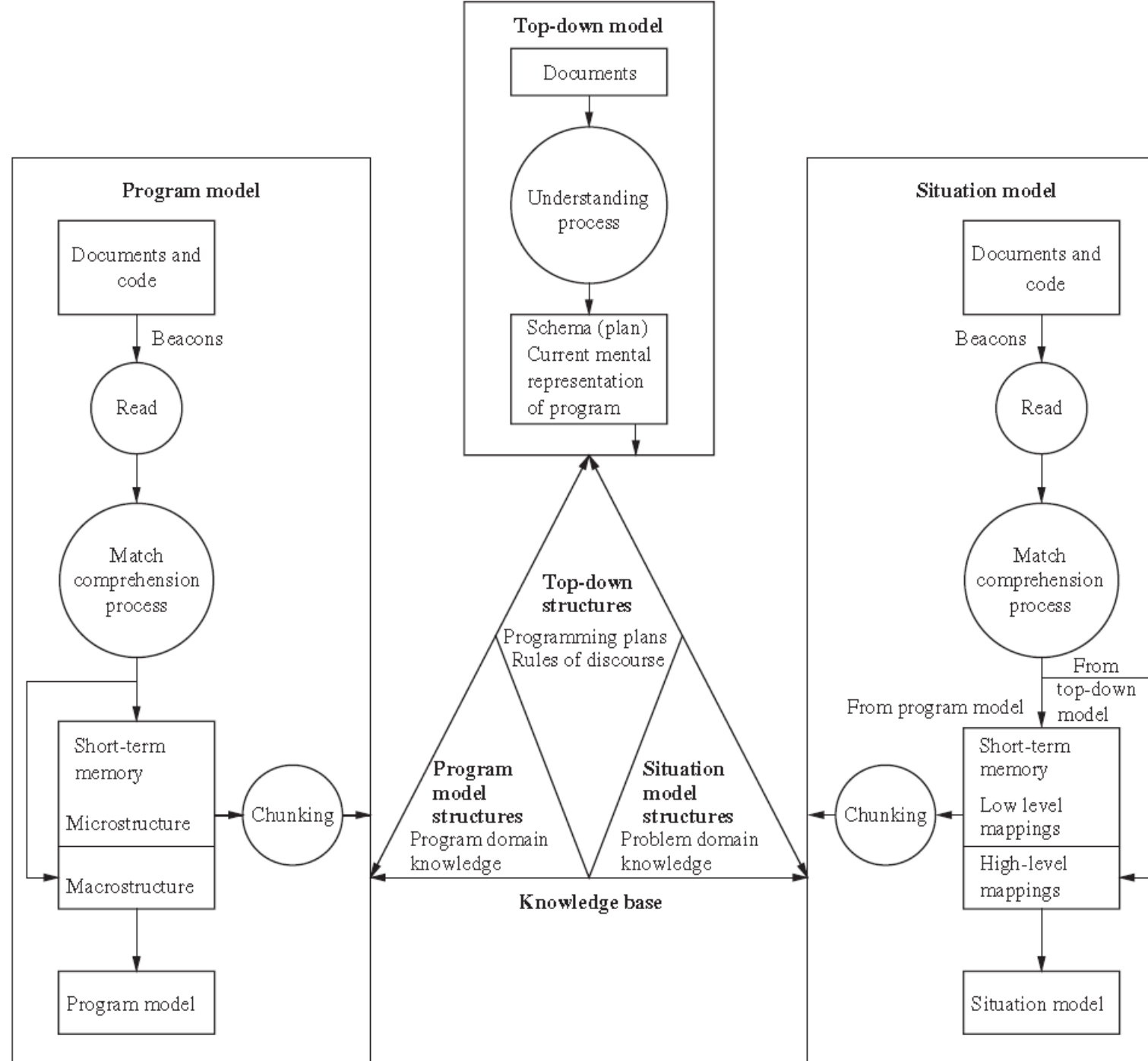






# Integrated Program Comprehension Model





# Questions

---

?

# Readings

---

□ Chapter 8: 8.1, 8.2 , 8.3