Chapter 6

Program Understanding

Overview

- Importance and aims of program comprehension
  - Program domain, Execution effect, Cause effect relation, Product Environment relation. Decision Support Features
- Maintainers and Their Information Needs
  - Managers, Analysts, Designers, and programmers
- Comprehension Process Models
  - Read the program, read the source code and run the program
- Mental Models
- Program Comprehension Strategies
  - Top-down
  - Bottom up
  - Opportunistic

Overview

- Factors that Affect Understanding
  - Expertise, Implementation issue, Documentation, Organization and Presentation of Programs, Comprehension support tools.

- Implications (insinuation) of Comprehension Theories and Studies
  - Knowledge acquisition, Education and Training, Design Principles, Guidelines and Recommendations

Program Comprehension

- Prior to implementing any change, it is essential to understand the software product as a whole and the programs affected by the change in particular

- Program comprehension is the process of reconstructing knowledge and relationship between problem and programming domains.

Importance of Program Comprehension

Knowledge	Importance
1. Problem domain	- To assist in the estimation of resources - To guide the choice of suitable algorithms, methodologies, tools and personnel
2. Execution effect	To determine whether or not a change did achieve the desired effect

Aims of Program Comprehension

Knowledge	Importance
3. Cause-effect relation	To establish the scope of a change, to predict potential ripple effects and to trace data flow and control flow
4. Product environment relation	To ascertain how changes in the product's environment affect the product and its underlying programs
5. Decision-support Features	To support technical and management decision-making features processes

Aims of Program Comprehension

The ultimate purpose of reading and comprehending programs is to be able successfully to implement requested changes.

1. Problem Domain:
    a. Selection of personnel with the appropriate level of expertise and skills
    b. To direct maintenance personnel in the choice of suitable algorithms, methodologies and tools
2. Execution Effect
    a. Maintenance personnel need to know (or be able to predict) what results the program will produce for a given input
    b. It can assist the maintenance personnel to determine whether an implemented change achieved the desired effect

Aims of Program Comprehension

3. Cause-Effect Relation:

- To reason about how components of a software product interact during execution.
- To predict the scope of a change and any knock-on effect that may arise from the change.
- To trace the flow of information through the program. The point in the program where there is an unusual interruption of this flow may signal the source of a bug

Aims of Program Comprehension

4. Product-Environment Relation:

- To predict how changes in environmental elements will affect the product in general and the underlying programs in particular

5. Decision-Support Features:

- Can guide maintenance personnel in technical and management decision-making processes like option analysis, decision making, budgeting and resource allocation
- To determine which components of the system require more resource for testing.

Maintainers and Their Information Needs

Maintenance team - managers, analysts, designers and programmers - all have different comprehension or information needs depending on the level at which they function

1. Managers:
    a. To make informed decisions
    b. To estimate the cost and duration of a major enhancement, change
2. Analysts:
    a. To determine the functional (FR) and non-functional requirements (NFR), and to establish the relationship between the system the elements of its environment.
    b. To determine the implications of change (using context diagrams) on the performance of a system.

Maintainers and Their Information Needs

3. Designers: The design process of a software system can take place at two levels:

- Architectural design results in the production of functional components, conceptual data structures and the interconnection between various components. High Level Structure of the software. Big picture

- Detailed design results in the detailed algorithms, data representations, data structures and interfaces between procedures or routines. Internal workings of each components. Small picture

Maintainers and Their Information Needs

3. Designers: During maintenance, the designer's need:

- To determine how enhancements could be accommodated by the architecture, data structures, data flow and control flow of the existing system
- To get a rough idea of the size of the job, the areas of the system that will be affected, and the knowledge and skills that will be needed by the programming team

Data flow diagrams (DFD), control flow diagrams (CFD), structure charts and hierarchy process input/output (HIPO) charts can help the designer

Maintainers and Their Information Needs

4. Programmers: Programmer needs to know the function of individual components of the system and their causal relation.

- To decide on whether to restructure or rewrite specific code segments;
- To predict more easily any knock-on effect when making changes that are likely to affect other parts of the system;
- To hypothesise the location and causes of error
- To determine the feasibility of proposed changes and notify management of any anticipated problems.

Key Activities of Program Comprehension

1. Code Reading
2. Documentation Review
3. Identifying Software Architecture
4. Control Flow Analysis
5. Data Flow Analysis
6. Static Analysis
7. Dynamic Analysis
8. Reverse Engineering
9. Mental Model Building
10. Tool Utilization
11. Code Refactoring for Clarity
12. Traceability Mapping
13. Collaborative Code Review

Static Analysis

- Analysis without executing the program.
- Based on source code, intermediate code, or bytecode.
- Performed early in development.

Key Activities of Static Analysis

- **Lexical Analysis** - Tokenizing source code into identifiers, literals, operators, etc.
- **Syntax Analysis** – Verifying grammar rules and structure (parse trees).
- **Semantic Analysis** – Checking for type correctness, scope resolution, and meaning.
- **Control Flow Analysis** - Identifying all possible paths through a program.
- **Data Flow Analysis** - Tracking variable usage (definitions, uses, lifetimes).
- **Dependency Analysis** – Discovering module, function, and variable interdependencies.
- **Style Checking** - Verifying adherence to coding standards and conventions.
- **Security Flaw Detection** - Identifying common security flaws (e.g., buffer overflows, injection)
- **Metrics Calculation -** calculating LOC, complexity

Dynamic Analysis

- Analysis during execution of the program.
- Observes actual behavior and performance.
- Requires input data and test environments.

Key Activities of Dynamic Analysis

- **Memory Leak Detection** - Identifies unused memory that is not released, leading to potential performance issues.
- **Profiling (CPU, memory usage)** - Measures resource usage (like CPU time and memory) to optimize performance.
- **Code Coverage** - Analyzes which parts of the code are executed during testing to ensure sufficient test coverage.
- **Runtime Error Detection** - Detects errors that occur during program execution, such as null pointer exceptions.
- **Behavior Monitoring** - Observes how the application behaves under different conditions to detect anomalies.
- **Execution Tracing** - Records the sequence of executed instructions or function calls to help in debugging and understanding flow

Comprehension Process Models

Three actions involved in the understanding of a program are: (1) reading about the program, (2) reading its source code, and (3) running it.

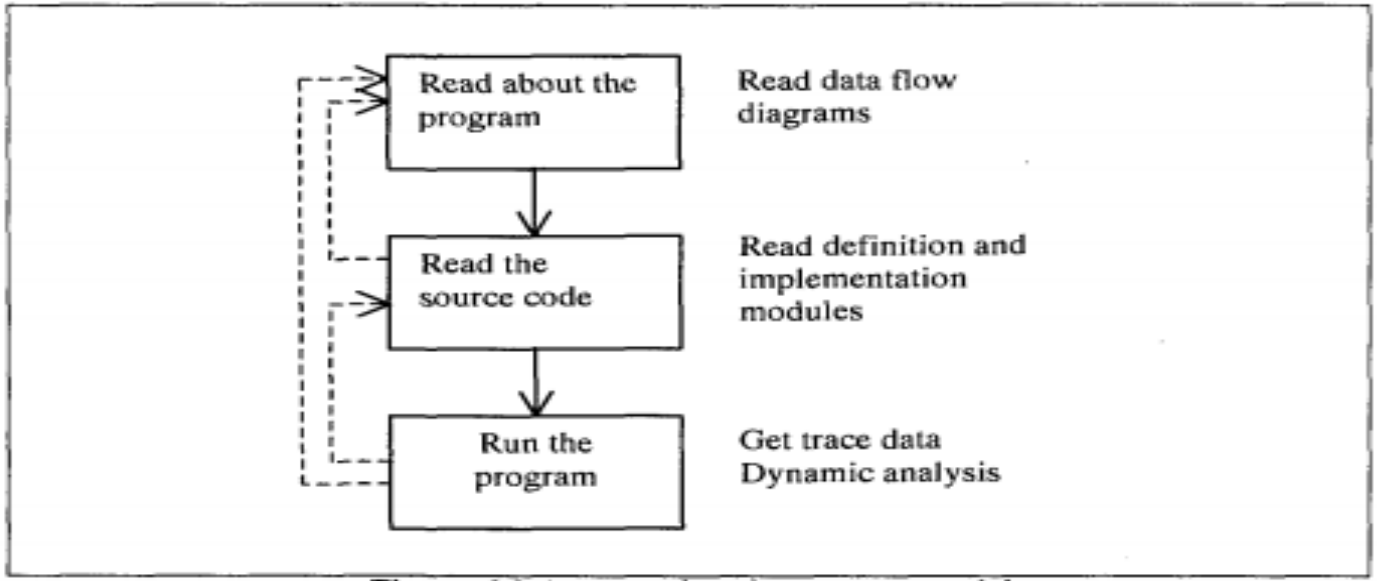


Figure 6.2 A comprehension process model

Comprehension Process Models

1. Read about the program:

The ‘understander' browses, peruses different sources of information such as the system documentation - specification and design documents - to develop an overview or overall understanding of the system.

If system information (for many large, complex and old systems) is not accurate, this phase may be omitted

Comprehension Process Models

2. Read the source code:

The global and local views of the program can be obtained in this phase.

- Global view is used to gain a top-level understanding of the system and to know effect a change might have on other parts of the system
- Local view allows programmers to have a low-level focus on a specific part of the system. It gives detail about system's structure, data types and algorithmic patterns

Tools such as static analysers - used to examine source code - are employed during this phase

Comprehension Process Models

3. Run the program:

The aim of this step is to study the dynamic behaviour of the program in action, including for example, executing the program and obtaining trace data.

The benefit of running the program is that it can reveal some characteristics of the system which are difficult to obtain by just reading the source code

Mental Models

Understanding how a system works, the behaviour of different parts, an algorithm, is known as the target system, and its mental representation is called a mental model.

Completeness and accuracy of the model depends to a large extent on its users' information needs.

Research in the area of programmers' cognitive behaviour during maintenance suggests that there are variations in the strategies that programmers use to understand programs (or form mental models of)

Program Comprehension

Def: Program comprehension is the process of reconstructing knowledge and relationship between problem and programming domains.

Software development in its entirety can be considered to be a design task which consists of two fundamental processes

1. **Composition**: It represents production of a design. Composition entails mapping what the program does in the problem domain, into a collection of computer instructions of how it works in the programming domain, using a programming language.

2. **Comprehension**: It is understanding the design produced by composition. It is a transformation from the programming domain to the problem domain involving the reconstruction of knowledge about these domains and the relationship between them.

Program Comprehension

It starts with a rough initial idea (called the primary hypothesis) about how the system works.

As more information is gathered from the code and documentation, this idea is confirmed and made more accurate.

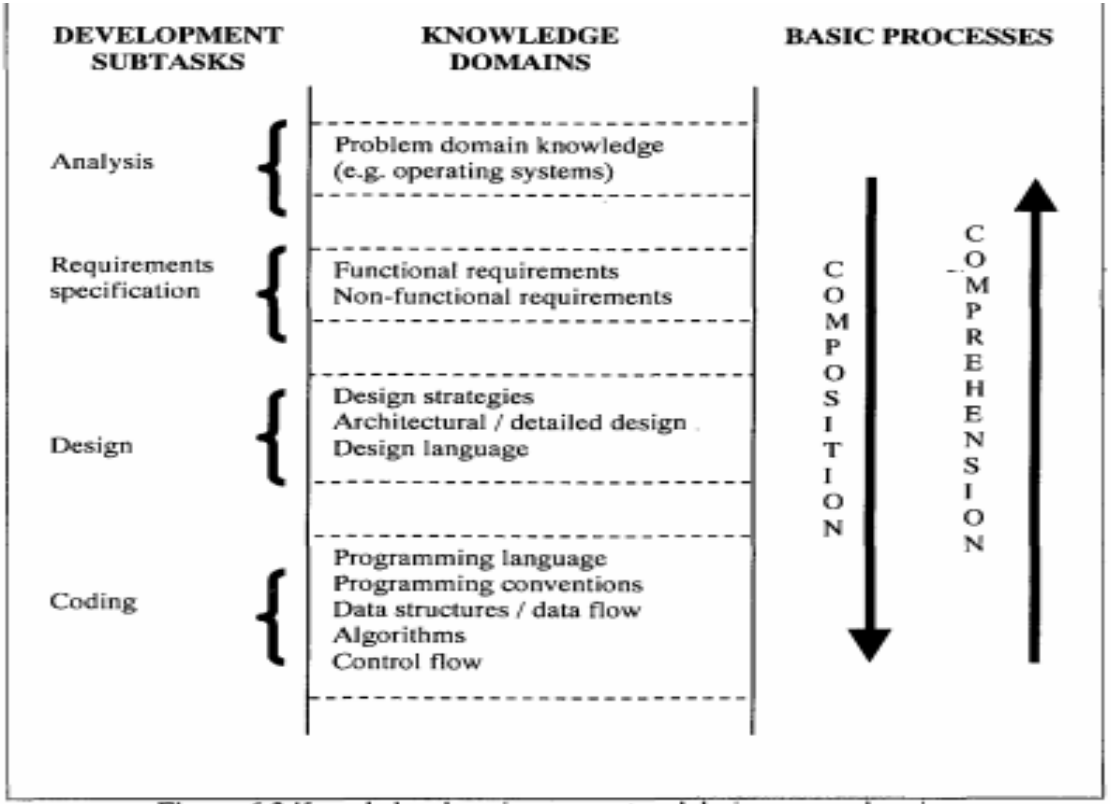


Figure 6.3 Knowledge domains encountered during comprehension

Beacons in Program Comprehension

The information required for hypothesis generation and refinement is manifested in key features - internal and external to the program - known as beacons.

No	Indicator
	Internal to the program text
1.	Prologue comments, including data and variable dictionaries
2.	Variable, structure, procedure and label names
3.	Declarations or data divisions
4.	Interline comments
5.	Indentation or pretty-printing
6.	Subroutine or module structure
7.	I/O formats, headers, and device or channel assignments
	External to the program
1.	Users' manuals
2.	Program logic manuals
3.	Flowcharts
4.	Cross-reference listings
5.	Published descriptions of algorithms or techniques

Program Comprehension Strategies

A program comprehension strategy is a technique used to form a mental model of the target program.

The mental model is constructed by combining information contained in the source code and documentation with the assistance of the expertise and domain knowledge that the programmer brings to the task.

Some models of how programmers go about understanding program are:

- Top-down
- Bottom-up
- Opportunistic

Program Comprehension Strategies

1. **Top-down Model:**

- **Tenet of the model:** You first try to understand what the program is supposed to do overall, then slowly dig into the details like how it works step by step, what kind of data it uses, and how everything flows.

- **Cognitive Process:** You start with a **hypothesis** about how the program works, then **check and improve** that guess as you learn more from the code and other sources.

- **Cognitive Structure:** You use your real-world knowledge, your programming knowledge, and build layers of understanding as you go deeper into the code.

Program Comprehension Strategies

2. Bottom-Up / Chunking Model:

- **Tenet of the model:** start by looking at **small parts** of the code, recognize familiar patterns, and then **combine them** to understand the bigger picture of what the program does.
- **Cognitive Process:** **group related pieces** of code together into high level semantic structure.
- **Cognitive Structure:** Mapping between knowledge domains.

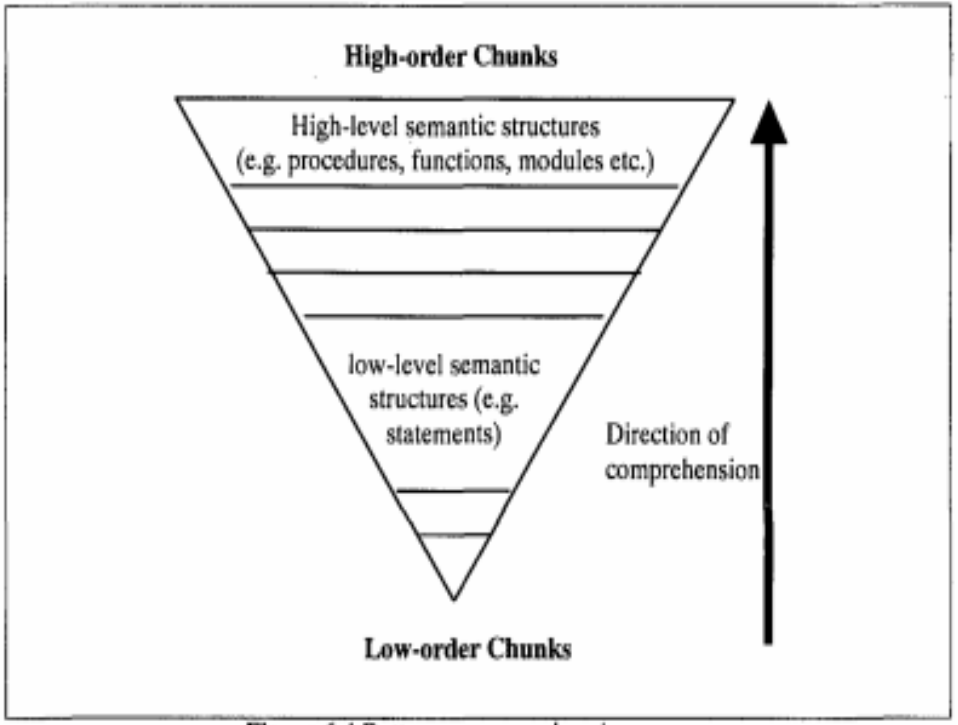


Figure 6.4 Bottom-up comprehension process

Program Comprehension Strategies

Weaknesses of both the Top-down and Bottom-up comprehension strategies are:

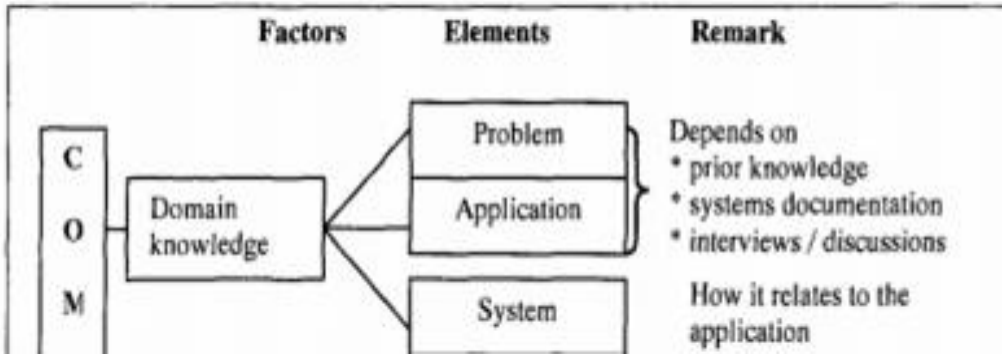
- Failure to take into consideration the contribution that other factors such as the available support tools make to understanding
- The fact that the process of understanding a program rarely takes place in such a well-defined fashion as these models portray. On the contrary, programmers tend to take advantage of any clues they come across in an opportunistic way.

Program Comprehension Strategies

3. Opportunistic Model:

- **Tenet of the model:** Understander makes use of both bottom-up and top-down strategies.
- According to this model, comprehension hinges on three key and complementary features - a knowledge base, a mental model and an assimilation process:
  - A knowledge base: This represents the expertise and background knowledge that the maintainer brings to the understanding task.
  - A mental model: This expresses the programmer's current understanding of the target program.
  - An assimilation process: This describes the procedure used to obtain information from various sources such as source code and system documentation.

Factors that Affect Understanding



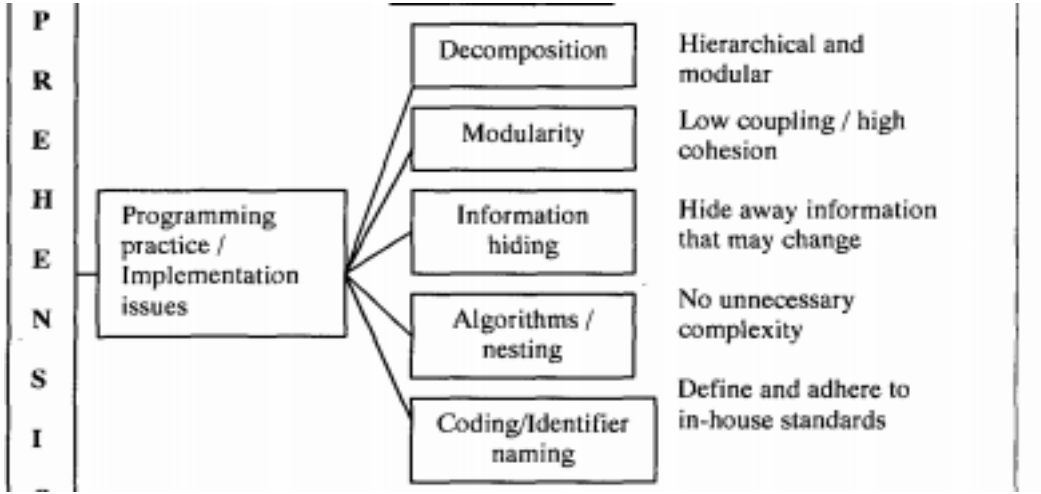
Following factors can affect formation of mental models of program, and their accuracy, correctness and completeness.

1. **Expertise:**
   There is a psychological argument that this expertise has a significant impact on comprehension. In many of these studies the performance of experts is compared with that of nonexperts (or novices). The expert tends to perform better than the novice.

Factors that Affect Understanding

2. **Implementation Issues:**
There are several implementation issues that can affect the ease and extent to which a maintainer understands a program

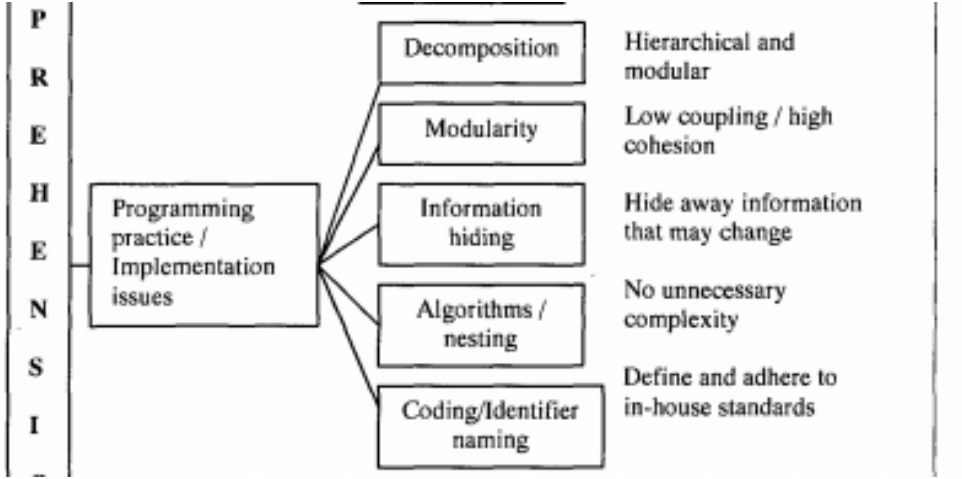
- **Naming Style:** Identifier names should be as informative, concise and unambiguous as possible.
- **Comments:** Programs with high-level comments were easier to modify. Comments in programs can be useful only if they provide additional information.



Factors that Affect Understanding

2. **Implementation Issues:**

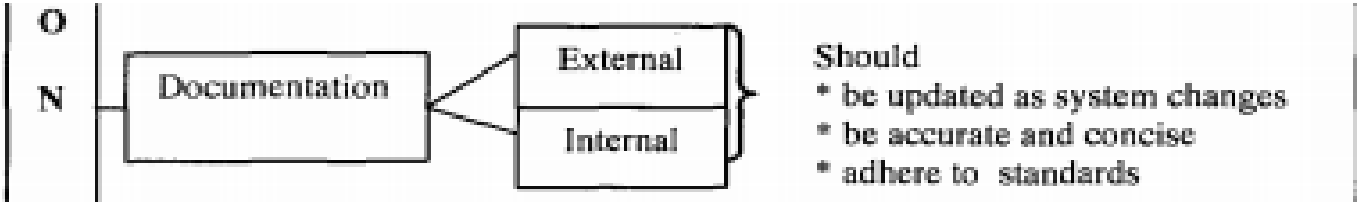
- **Decomposition Mechanism:** Comprehensibility depends to some extent on how the entire software system has been decomposed
  - **Modular decomposition:** It is a a technique for dividing large software systems into manageable components - called modules
  - **Structured programming**: Approach of using high-level programming languages to reduce the size and complexity of programs.



Factors that Affect Understanding

3. **Documentation**: Due to the high turnover of staff within the software industry, it is essential to maintain proper documentation.

Maintainers need to have access to the system documentation to enable them to understand the functionality, design, implementation and other issues.

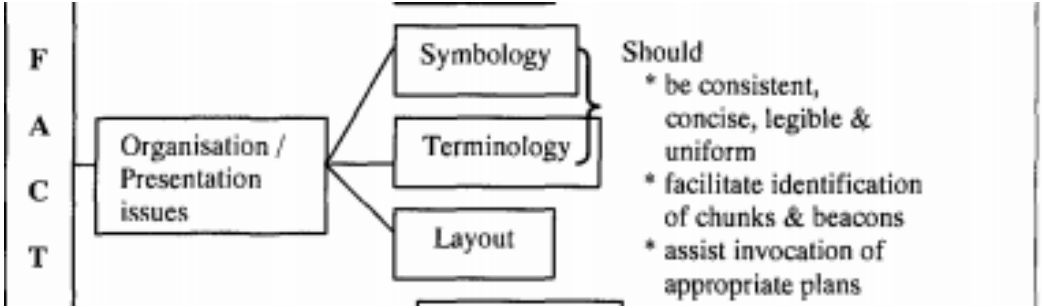


Factors that Affect Understanding

4. **Organisation and Presentation of Programs**:

Enhanced program presentation can improve understanding by

- Facilitating a clear and correct expression of the mental model of the program
- Emphasising the control flow the program's hierarchic structure and the programmer's - logical and syntactic - intent
- Visually enhancing the source code through the use of
  - Indentation
  - Spacing
  - Boxing and shading



Factors that Affect Understanding

4. Organisation and Presentation of Programs:

B: Blank lines / indentation / boxes / shading

```
FOR i:=1 TO NumEmployees DO
LowPos:=i
Smallest:=Employees[LowPos].EmployeeAge;

    FOR j:=i+1 TO NumEmployees DO

        IF Employee[j].EmployeeAge<Smallest THEN
            LowPos:=j;
            Smallest:=Employee[j].Employee.Age
        END (*IF*)

    END (* FOR j *)

    TempRec:=Employee[LowPos];
    Employee[SmallPos]:-Employee[i];
    Employee[i]:=TempRec

END (* FOR i *)
```

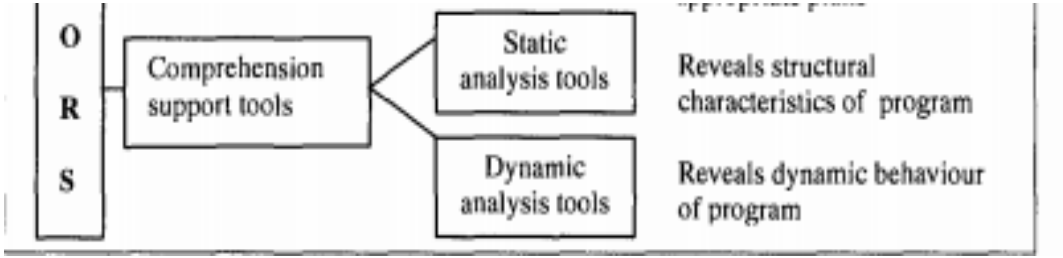
Figure 6.8 Blank lines, indentation, boxes and shading to improve program layout

Factors that Affect Understanding

5. **Comprehension Support Tools**:

There are tools which can be used to organise and present source code in a way that makes it more legible, more readable and hence more understandable.

Example: “Book Paradigm” involves documenting source code using publishing features and style traditionally found in books.



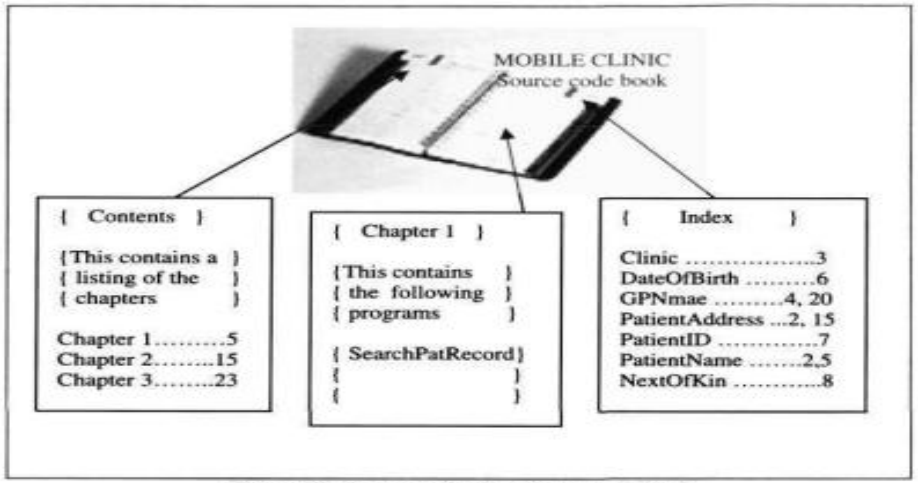


Figure 6.9 **An example of a source code book**

Implications of Comprehension Theories and Studies

1. Knowledge Acquisition and Performance:

The knowledge that a maintainer requires for modifying a program depends on the nature of the change.
The maintainers who used a systematic strategy for program comprehension (similar to the opportunistic) were successful in making modifications because they gathered knowledge about the cause effect relation of the system's functional units.

2.Education and Training:

Maintainers need to be taught about program understanding. They can reflect on the appropriateness of the strategy that they usually use.

Implications of Comprehension Theories and Studies

3. Design Principles:

Comprehension hinges on the ability to form an accurate mental model of the target program. Lessons can be learnt about appropriate principles for designing programs, programming languages, documentation standards and support tools that facilitate the formation of mental models.

4.Guidelines and Recommendations:

Results from empirical studies provide a basis for software maintainers to set guidelines for programming and documentation practices.