CHAPTER 5
Measuring internal product attributes: size

- Software size
- Software Size: Length (code, specification, design)
- Software Size: Reuse
- Software Size: Functionality (function point, feature point, object point, use-case point)
- Software Size: Complexity

Measuring Internal Product Attributes: Size

Definition

- **Size** is one of the most fundamental **internal product attributes** of software.
  It quantifies **how large or complex a software product is**, based on measurable internal properties such as **lines of code, number of components, functions, or data items**.
- **Internal attributes** are those that can be measured **directly from the product itself**, without executing it — unlike external attributes (e.g., reliability, usability, performance).

Purpose of Measuring Size

- Measuring software size helps to:
- Estimate **development effort**, **cost**, and **schedule**.
- Assess **productivity** (e.g., lines of code per person-month).
- Support **quality assurance** and **project control**.
- Enable **comparison** between systems or modules.

Software Size

Definition

- **Software size** is a quantitative measure that indicates **how big a software system or its components are**.
  It represents the **amount of functionality, code, or logic** contained in a software product.
- In simple terms, it tells us *"how much software has been developed."*

Purpose of Measuring Software Size

- Measuring software size helps in:
- **Effort and cost estimation** — larger software usually needs more time and resources.
- **Project planning** — deciding manpower, schedule, and milestones.
- **Productivity measurement** — e.g., LOC per person-month.
- **Quality assurance** — identifying modules that are too large or complex.
- **Maintenance management** — tracking growth or changes in the product.

Main Categories of Software Size Measures

There are two major ways to measure software size:

1. Physical Size

- This method measures the **internal structure** of the software.

Common metrics:

- **Lines of Code (LOC):**
  Counts the total number of lines in the program (can be physical or logical lines).
  Example: 2,000 LOC.
- **Number of Modules / Classes / Functions:**
  Counts structural elements of the program.

Advantages:

- Easy to calculate once code is available.
- Directly related to programming effort.

Disadvantages:

- Depends on programming language and coding style.
- Cannot be used before coding starts.

2. Functional Size

- This method measures the **functionality delivered to the user**, independent of how it is coded.

Common metric:

- **Function Points (FP):**
  Counts features such as inputs, outputs, user interactions, and files.
  The total is adjusted based on complexity to estimate overall size.

Advantages:

- Can be used early in development (before coding).
- Language-independent and user-oriented.

Disadvantages:

- Requires detailed system analysis.
- Some subjectivity in judgment of complexity.

Typical Software Size Metrics

Metric	Description	Usage
LOC (Lines of Code)	**Number of code lines in source files**	**Implementation and testing stages**
Function Points (FP)	**Functional capabilities offered to user**	**Early project estimation**
Object Points	**Counts UI screens, reports, and modules**	**GUI or 4GL systems**
Module Count	**Number of modules or components**	**Structural analysis**

Software Size: Length

Definition

- **Software length** refers to the **physical size** of the software system measured by the **amount of code written**.
  It is usually expressed as the **number of program statements or lines of code (LOC)** in the software product.
    - Software length = *How long the source code is.*

Purpose of Measuring Length

- Measuring software length helps in:
- Estimating **effort**, **cost**, and **time** required for development.
- Evaluating **productivity** (e.g., LOC per person-month).
- Comparing **different versions** of the same software.
- Assessing **maintainability** — shorter, modular code is easier to maintain.

Common Length Metrics

- The most widely used metric for length is **Lines of Code (LOC)**. There are two main types:
- **Physical Lines of Code (PLOC):**
  Counts every line in the source file, including comments and blanks (sometimes excluded based on convention).

Example:

```
int a, b;
a = b + 5;
printf("Result: %d", a);
```

➢3 physical lines of code.

- **Logical Lines of Code (LLOC):**
  Counts **only executable statements** that perform operations or actions.
  Comments and blank lines are not counted.

- Length-Based Metrics Examples

Metric	Meaning	Example Value
Total LOC	Total number of lines in a program	8,500 LOC
LOC per Module	Average length of modules	425 LOC/module
LOC per Function	Average lines per function	40 LOC/function

These measurements can be used for productivity or complexity evaluation.

Advantages:

- Simple and direct to calculate.
- Useful for comparing different programs written in the **same language**.
- Forms the basis for other metrics like **defect density** (defects/LOC) or **productivity** (LOC/person-month).

Disadvantages:

- **Language-dependent** – 1 line in C ≠ 1 line in Java or Python.
- **Coding style variation** – Different programmers may write the same logic in varying numbers of lines.
- **Not meaningful early in the lifecycle** – Cannot measure until coding is done.

Software Size: Reuse

Definition

- **Software size (reuse)** refers to the **portion of a software system that is derived from existing components** rather than newly developed code.
- In other words, it measures **how much of the software product's size** comes from **reused modules, libraries, or code** instead of being written from scratch.
- It represents the *extent to which existing software assets are reused* to build a new system.

Purpose of Measuring Reuse

- Measuring software reuse helps organizations to:
- **Estimate effort and cost** more accurately (reused code reduces development effort).
- **Assess productivity** by recognizing savings from reuse.
- **Evaluate quality** — reused components are often more reliable.
- **Plan maintenance** — reused parts may depend on external libraries or systems.
- **Track progress** toward software reuse goals or policies.

Components of Reuse Measurement

- Software systems typically consist of both **new** and **reused** code. Hence, total software size can be expressed as:

  Total Size=New Code+Reused Code+Modified Reused Code

Where:

- **New Code:** Developed from scratch.
- **Reused Code:** Integrated without modification.
- **Modified Reused Code:** Existing code changed for adaptation.

Benefits of Reuse

- **Reduces development effort and cost**
- **Increases reliability** (reused code is already tested)
- **Improves productivity**
- **Speeds up delivery**
- **Encourages standardization and modular design**

Common Reuse Size Metrics:

Metric	Description	Formula / Example
Percentage of Reuse	Measures proportion of reused code in total system	$\text{Reuse } \% = \frac{\text{Reused LOC}}{\text{Total LOC}} \times 100$
Reuse Ratio	Ratio between reused and new code	$\text{Reuse Ratio} = \frac{\text{Reused LOC}}{\text{New LOC}}$
Reused Function Points (RFP)	Portion of function points derived from reused components	Used in Function Point-based estimation models

Example

Suppose a project has:

- Total size = 10,000 LOC
- 6,000 LOC reused from previous projects
- 4,000 LOC newly developed

Then:

$$\text{Reuse } \% = \frac{6,000}{10,000} \times 100 = 60\%$$

This means **60% of the software** was developed using **existing components**, significantly reducing effort and time.

Limitations

- Reused components may need **adaptation** to fit new requirements.
- **Integration issues** can occur between old and new modules.
- Reuse may **propagate existing bugs**.
- **Dependency management** becomes critical.

Software Size: Functionality

Definition

- **Software functionality size** refers to the **amount of useful work or services** that a software system provides to its users.
  It measures **what the software does** rather than **how it is implemented**.
  - Software functionality = *the total functions, features, and operations a system performs for users.*

  This approach focuses on **user requirements** and **system behavior**, not lines of code.

Purpose of Measuring Functionality

- Measuring software functionality helps to:
- Estimate **development effort and cost** early in the lifecycle (before coding).
- Evaluate **productivity** based on the amount of functionality delivered.
- Compare systems built in **different languages or technologies**.
- Assess **software quality** and **user satisfaction** by function coverage.

Characteristics of Functional Size

Functional size is:

- **Language-independent** — applicable to any programming language.
- **Technology-independent** — focuses on logic and behavior, not code.
- **User-oriented** — describes what users can do with the system.

Common Functional Size Measurement Methods

a) Function Point Analysis (FPA)

- The most widely used method to measure functional size.
- It counts the **number and complexity** of user-visible functions such as:
- External Inputs (EI)
- External Outputs (EO)
- External Inquiries (EQ)
- Internal Logical Files (ILF)
- External Interface Files (EIF)
- Each function is weighted (simple, average, complex), and total **Unadjusted Function Points (UFP)** are computed.
  Then, complexity adjustment factors (CAF) are applied.

$$\text{Function Points (FP)} = \text{UFP} \times (0.65 + 0.01 \times \text{CAF})$$

b) Feature Points

- An extension of Function Points that also includes **algorithmic complexity** — useful for scientific or engineering software.

c) Use Case Points

- Based on **use cases** from UML models — counts actors and use case complexity.

Advantages:

- **Language-independent** measure.
- Focused on **user-visible functionality**.

Disadvatages:

- Requires detailed analysis of requirements or design.
- Can be subjective if complexity factors are misestimated.

Example:

- A payroll system:
- 20 external inputs, 10 outputs, 5 files → 150 Function Points (FP)

Software Size: Complexity

Definition

- **Software complexity** refers to the **degree of difficulty involved in understanding, developing, testing, and maintaining** a software system or its components.
  It reflects how **complicated the control flow, data flow, or structure** of the code is.
- In terms of size, **software complexity** indicates the **"intellectual size"** of the software — i.e., how large or difficult the system is to comprehend, not just how many lines it has.

Software Size (Complexity) measures *how hard the software is to understand or work with*, rather than how big it is physically.

Purpose of Measuring Complexity

- Measuring complexity helps in:
- **Estimating effort** — complex programs take more time to develop and test.
- **Predicting defects** — higher complexity often leads to more errors.
- **Improving maintainability** — identifying modules that need simplification.
- **Enhancing code quality** — promoting modular and structured design.
- **Assessing reliability** — simpler designs are easier to verify and validate.

Metrics:

- **Cyclomatic Complexity (McCabe)**: Number of independent paths through code.
- **Halstead Metrics**: Based on operators, operands, and program vocabulary.
- **Object-Oriented Metrics**: Depth of inheritance, coupling, cohesion.

Cyclomatic Complexity (McCabe)

- A very popular measure of logical complexity.

$$V(G)=E-N+2P$$

Where:

- E = number of edges (control flow transfers)
- N = number of nodes (statements/blocks)
- P = number of connected components (usually 1 for a single program)

Example:
If a function contains 10 decision points (if/else/while),
then V(G)=10+1=11

→ **Cyclomatic Complexity = 11**

Advantages of Measuring Complexity

- Identifies **risky modules** early.
- Supports **better testing** (e.g., number of paths to test).
- Aids **refactoring** and code optimization.
- Helps maintain **software quality standards**.

Limitations

- Complexity metrics may not capture all real-world difficulties.
- Some metrics are **mathematical but not intuitive** for developers.
- May vary across languages or paradigms.