Chapter 23: Product Metrics

Software Engineering

BTech Computer Science and Engineering
1 Hour

Introduction to Product Metrics

Measuring Software Quality Objectively

The Measurement Challenge

The Core Question: How do you know if your software design is actually "good"?

Moving Beyond Intuition: We cannot rely on guesswork or subjective opinions to evaluate software.

What Are Product Metrics?

Quantitative, objective measures of software characteristics.

Used to evaluate size, complexity, maintainability, and testability.

The Value of Metrics

Measurement allows us to predict outcomes, control processes, and continuously improve the final product.

Lecture Objectives

Mastering Software Measurement

By the end of this lecture, you will be able to:

- Define the core concepts: measures, metrics, and indicators, while understanding foundational measurement principles.
- Apply goal-oriented software measurement using the Goal/Question/Metric (GQM) paradigm.
- Calculate function-based metrics (like Function Points) directly from the requirements model.
- Evaluate the quality of a specification using objective metrics.
- Apply specific architectural design metrics and object-oriented design metrics to your projects.

Measures, Metrics, and Indicators

Understanding the Hierarchy of Data

Measure

A raw, quantitative value obtained directly through measurement (e.g., 100 lines of code).

Metric

A quantitative measure representing the degree to which a system possesses a specific attribute (e.g., Cyclomatic complexity of 15).

Indicator

A metric (or combination of metrics) that provides deep insight into the software process or product itself.

Example Progression:

Measure: 500 lines of code → **Metric:** Cyclomatic complexity is 25 → **Indicator:** "High complexity"

The Challenge of Product Metrics

Measuring the Intangible

How do we accurately measure intangible qualities like "maintainability" or "reusability"?

Key Challenges:

- **Conceptual Nature**: Software is logical and conceptual, not a physical object you can put on a scale.

- **Lack of Standard Units**: There is no universal "meter" or "kilogram" for software quality.

- **Validation Needs**: Metrics must be strictly validated (does this number actually measure what it claims to?).

- **Risk of Gamification**: Metrics can be misused or artificially "gamed" by developers to look good on paper.

Measurement Principles

Guidelines for Collecting Data

- **Establish Objectives First:** Know exactly why you are measuring before collecting any data.
- **Define Unambiguously:** Every metric requires a crystal-clear operational definition.
- **Align with Priorities:** Metrics must be derived directly from the organization's broader goals. Do not measure just for the sake of measuring.
- **Integrate into Workflows:** Collect data seamlessly as part of the normal development process, not as a burdensome afterthought.
- **Contextualize:** Always interpret metrics in context (e.g., a high complexity number might be perfectly acceptable in a highly mathematical domain).

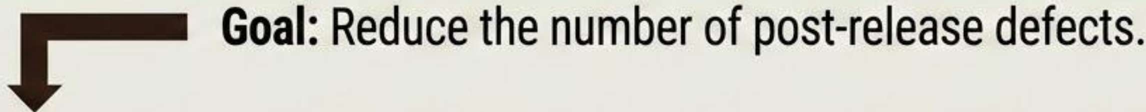
The GQM Paradigm

Goal-Oriented Software Measurement

The Goal-Question-Metric (GQM) Approach

- **Goal:** State the overarching measurement objective.
- **Question:** Ask specific questions that determine if the goal is being met.
- **Metric:** Define the exact quantitative metrics needed to answer those questions.

GQM Example in Action



Question 1: How effective are our code reviews? → **Metric 1:** Defects found per review hour.

Question 2: How complex is our code? → **Metric 2:** Cyclomatic complexity per module.

Question 3: How well are we testing? → **Metric 3:** Statement coverage percentage.

Attributes of Effective Metrics

Evaluating Our Measurement Tools

Simple and Computable: Easy to calculate automatically, without requiring massive manual analysis.

Objective: Based on hard math, not open to subjective human interpretation.

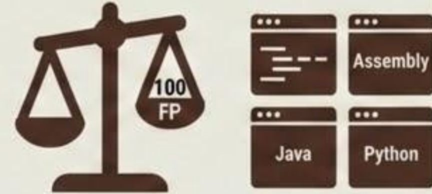
Empirically Validated: Scientifically shown to actually correlate with real-world quality attributes.

Consistent: Yielding the exact same result for the same artifact, regardless of who is doing the measuring.

Economical: The financial and time cost of collecting the metric must be justified by the business benefit it provides.

Function-Based Metrics

Measuring Size by Functionality, Not Code



Origin:

Developed by Allan Albrecht at IBM in 1979.

The Purpose:

To measure the size of a software project based on the actual functionality delivered to the user, rather than counting Lines of Code (LOC).

The Big Advantage:

Function Points are entirely independent of the programming language. (100 Function Points is the same size project whether you code it in Assembly, Java, or Python).

Components of Function Points

The Building Blocks of an Application

The 5 Information Domain Values

External Inputs (EI)

User inputs that provide application-oriented data (e.g., submitting a user registration form).

External Outputs (EO)

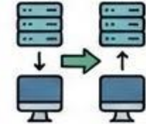
Reports or screens generated by the application for the user (e.g., displaying a dashboard or printing a receipt).

External Inquiries (EQ)

Interactive queries that retrieve data without altering it (a combination of a simple input and an immediate output).

Internal Logical Files (ILF)

Logical data stores maintained within the application (e.g., internal database tables).



External Interface Files (EIF)

Data stores accessed by the application but maintained by another system (e.g., a third-party payment gateway database).

Function Point Calculation

Weighting and Adjusting the Score

Step 1: Unadjusted Function Points (UFP)

- Each component is rated as Simple, Average, or Complex.
- We multiply the count of each component by a standardized weight (e.g., a Complex ILF is worth 15 points; a Simple EI is worth 3 points) and sum them up.

Step 2: Value Adjustment Factor (VAF)

- We rate 14 general system characteristics (like performance needs, reusability, distributed processing) on a scale of 0 to 5.

Formula:
VAF = 0.65 + (sum of ratings / 100)

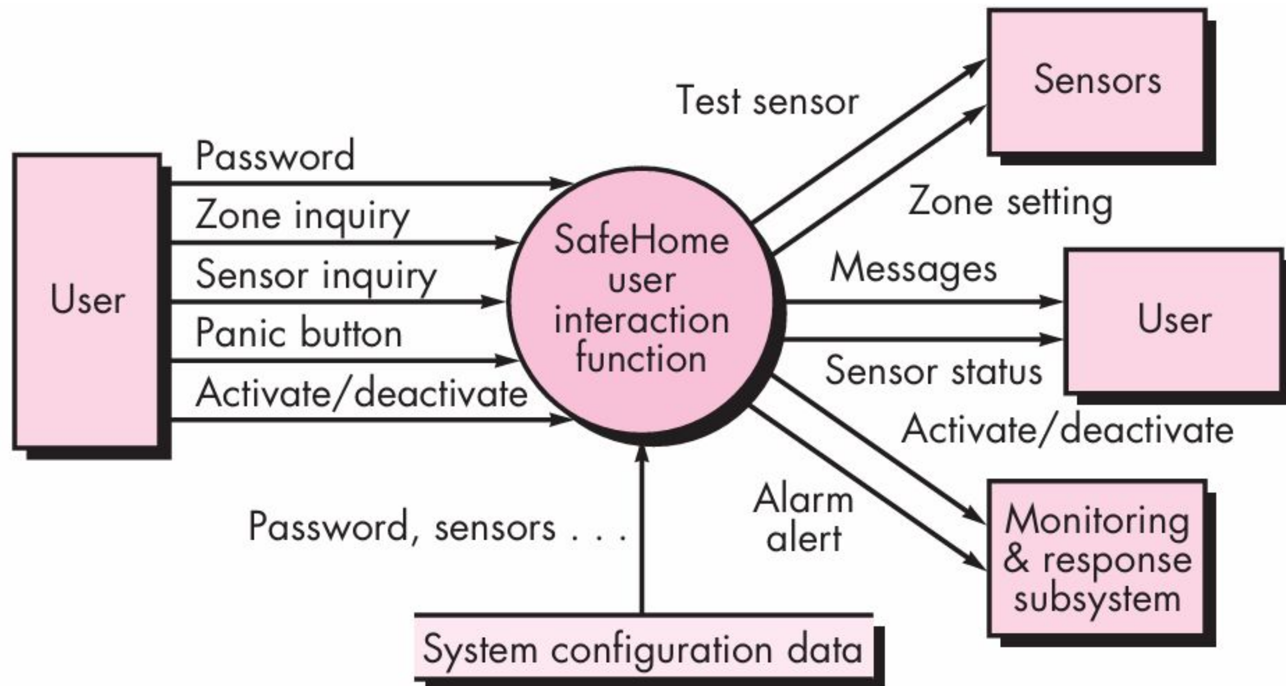
Step 3: Adjusted Function Points (AFP)

Formula:
AFP = UFP × VAF

The final size measurement for the software project.

FIGURE 23.2

A data flow model for *SafeHome* software



To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure 23.2. Referring to the figure, a data flow diagram (Chapter 7) for a function within the *SafeHome* software is represented.

FIGURE 23.1

Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total →							

The data flow diagram is evaluated to determine a set of key information domain measures required for computation of the function point metric. Three external inputs—**password, panic button**, and **activate/deactivate**—are shown in the figure along with two external inquiries—**zone inquiry** and **sensor inquiry**. One ILF (**system configuration file**) is shown. Two external outputs (**messages** and **sensor status**) and four EIFs (**test sensor, zone setting, activate/deactivate,** and **alarm alert**) are also present. These data, along with the appropriate complexity, are shown in Figure 23.3.

The count total shown in Figure 23.3 must be adjusted using Equation (23.1). For the purposes of this example, we assume that $\Sigma(F_i)$ is 46 (a moderately complex product). Therefore,

$$FP = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Based on the projected FP value derived from the requirements model, the project team can estimate the overall implemented size of the *SafeHome* user interaction

FIGURE 23.3

Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)	3	×	(3)	4	6	=	9
External Outputs (EOs)	2	×	(4)	5	7	=	8
External Inquiries (EQs)	2	×	(3)	4	6	=	6
Internal Logical Files (ILFs)	1	×	(7)	10	15	=	7
External Interface Files (EIFs)	4	×	(5)	7	10	=	20
Count total						→	50

Using Function Points

Turning Size into Actionable Business Metrics

Once the Adjusted Function Point (AFP) total is calculated, it becomes the baseline for critical business metrics:

Productivity

$$\frac{\textbf{Function Points}}{\textbf{Person-months}}$$

(How fast is our team actually delivering business value?)

Quality

$$\frac{\textbf{Defects}}{\textbf{Function Point}}$$

(How many bugs are we creating per unit of delivered functionality?)

Cost Efficiency

$$\frac{\textbf{Project Cost}}{\textbf{Function Points}}$$

(How much does it cost us to build one unit of software functionality?)

Metrics for Specification Quality

Measuring the Quality of the Blueprint

The Goal: Measure the quality of the requirements specification document before design or coding begins.

Key Metrics

- **Defects per page:** Number of errors found in review per page. (Indicates raw specification quality).
- **Specification volatility:** Requirements Changes / Total Requirements. (Indicates the stability of the project).
- **Ambiguity count:** Number of vague or ambiguous statements per page. (Indicates clarity).
- **Traceability:** Percentage of requirements successfully traced to design/code. (Indicates completeness).

The Quality Indicator

A specification with high defect density or high volatility acts as a massive warning sign for costly rework later in the lifecycle.

Metrics for the Design Model

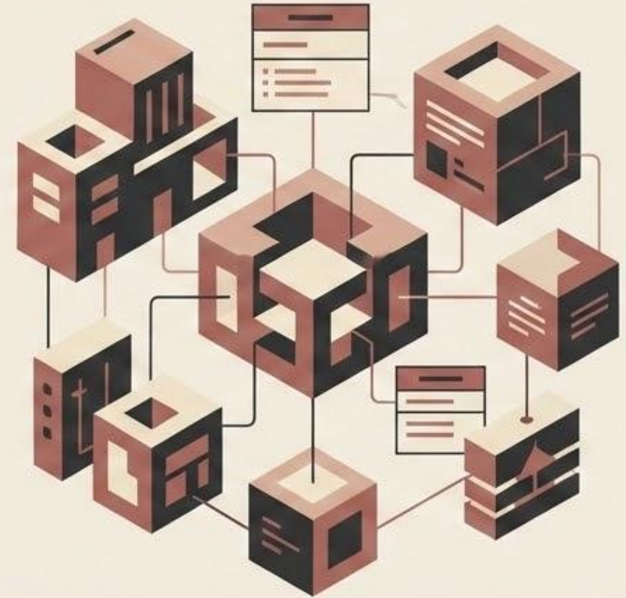
Evaluating Architecture and Object-Oriented Quality

The Purpose

To quantitatively evaluate the quality of a software architecture before extensive coding begins.

Key Focus Areas

- **Architectural Design Metrics:** Assessing the macro-level structure, modularity, and control flow of the system.
- **Object-Oriented (OO) Design Metrics:** Measuring specific quality attributes of classes and objects, such as maintainability, reusability, and testability.



Architectural Design Metrics

Structural Complexity (Henry & Kafura)

Module Complexity (MC)

Evaluates complexity based on the flow of information into and out of a module.

- **Fan-in**: The number of distinct modules that call (depend on) this specific module.
- **Fan-out**: The number of distinct modules that this specific module calls.

The Formula:

$$Complexity = Fan\text{-}out \times (Fan\text{-}in + Fan\text{-}out)$$

Interpretation:

- A high complexity score strongly suggests poor modularity.
- Modules with high complexity are notorious bottlenecks for future maintenance and testing.

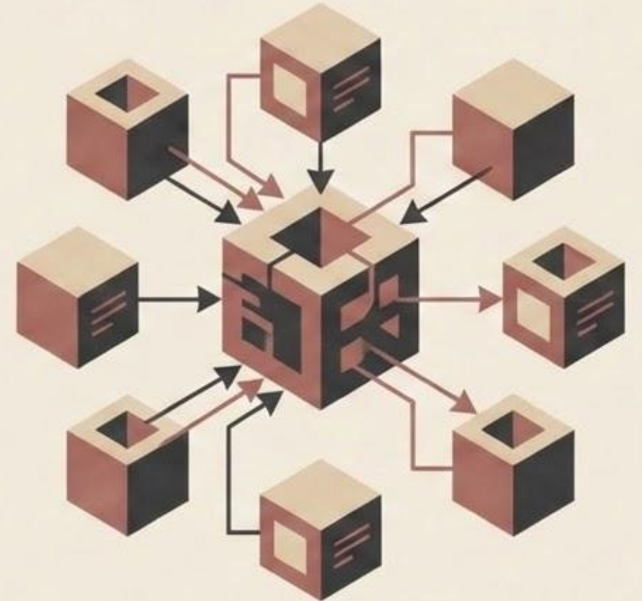
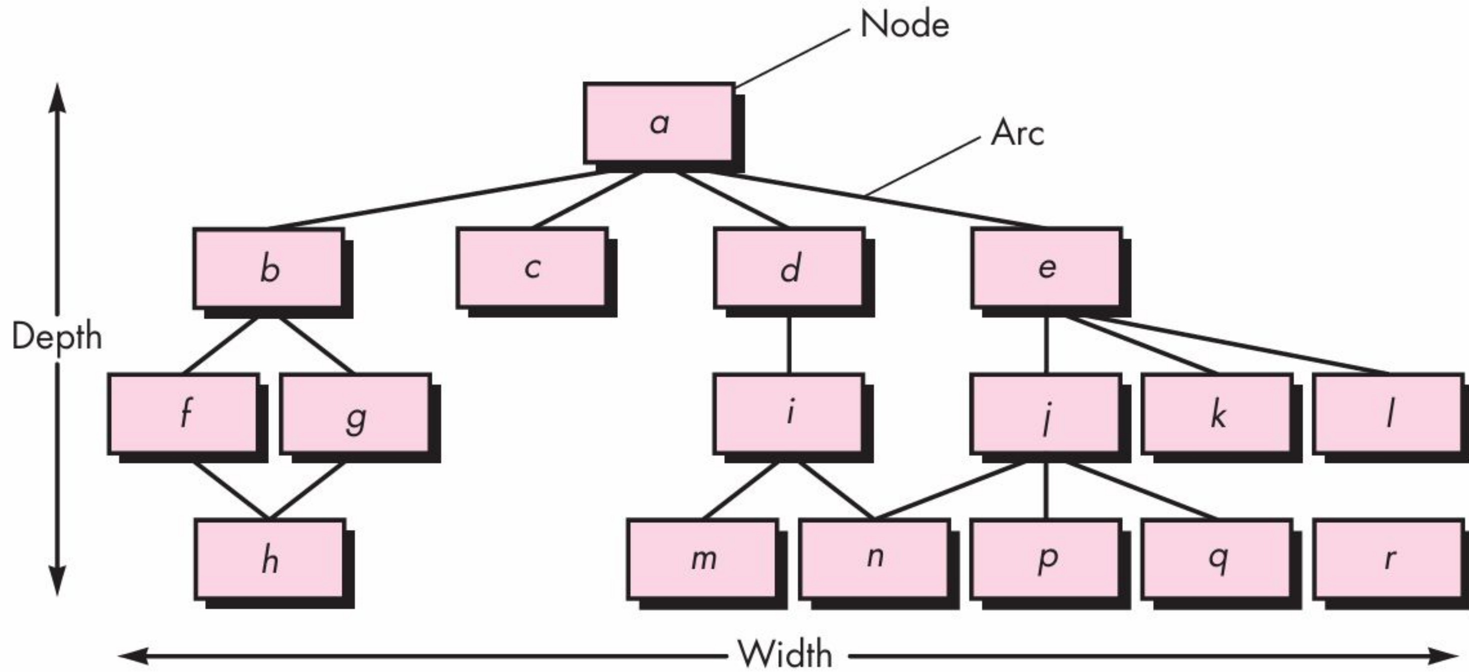


FIGURE 23.4

Morphology metrics



Architecture Quality Indicators

The Hallmarks of Good Design

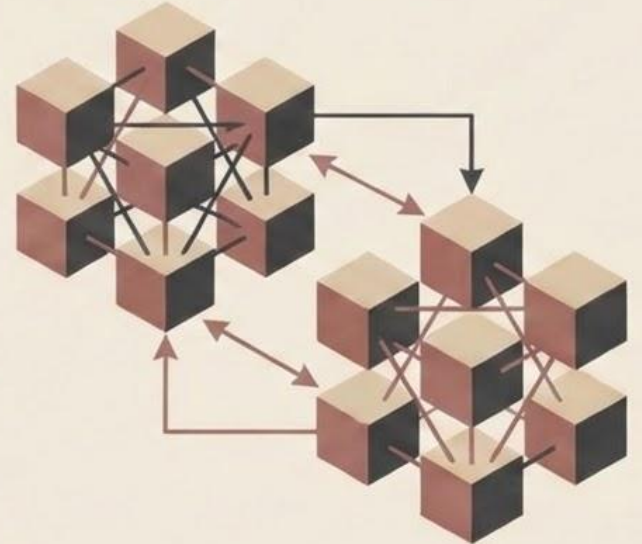
Coupling and Cohesion

- **Coupling Between Modules (CBM)**: The absolute count of inter-module connections.
- **Cohesion of a Module (CM)**: An assessment of how well the internal elements of a single module work together toward a single purpose.

The Golden Rule: Low Coupling + High Cohesion = Good Design.

Architecture Quality Indicators

- **Cyclomatic Complexity**: Measures the number of independent paths/decision points in the control structure.
- **Depth of Inheritance Tree (DIT)**: The maximum number of ancestor classes for an object.
- **Number of Children (NOC)**: The number of immediate subclasses derived from a parent class.



Object-Oriented Design Metrics

The CK Metrics Suite (Chidamber and Kemerer (1994)

The CK Suite is the industry standard for measuring Object-Oriented design quality:

WMC (Weighted Methods per Class)

- **Definition:** Sum of the complexities of all methods in a single class.
- **Interpretation of High Values**: Extremely difficult to maintain or reuse.

DIT (Depth of Inheritance Tree)

- **Definition**: Maximum path length from the class to the root class.
- **Interpretation of High Values**: High reuse of inherited behavior, but heavily complex.

NOC (Number of Children)

- **Definition**: Number of immediate subclasses attached to a class.
- **Interpretation of High Values**: High potential for reuse, but requires rigorous testing.

CBO (Coupling Between Objects)

- **Definition**: Number of other independent classes this class is coupled to.
- **Interpretation of High Values**: Highly sensitive to changes in other parts of the system.

RFC (Response for a Class)

- **Definition**: Total number of methods that can be invoked in response to a message.
- **Interpretation of High Values**: Highly complex testing and debugging required.

LCOM (Lack of Cohesion of Methods)

- **Definition**: Number of method pairs within a class that do not share attributes.
- **Interpretation of High Values**: The class is doing too many different things (poor cohesion).

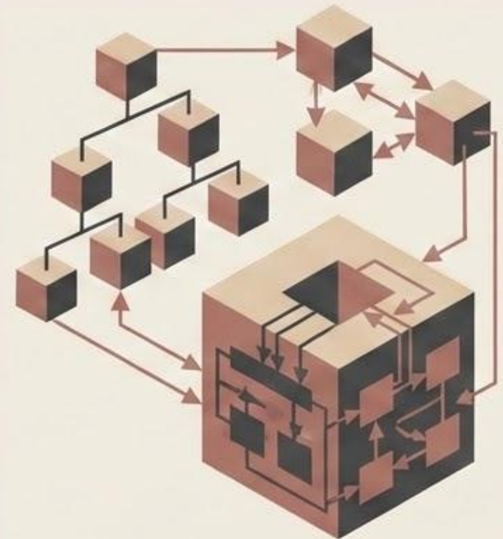
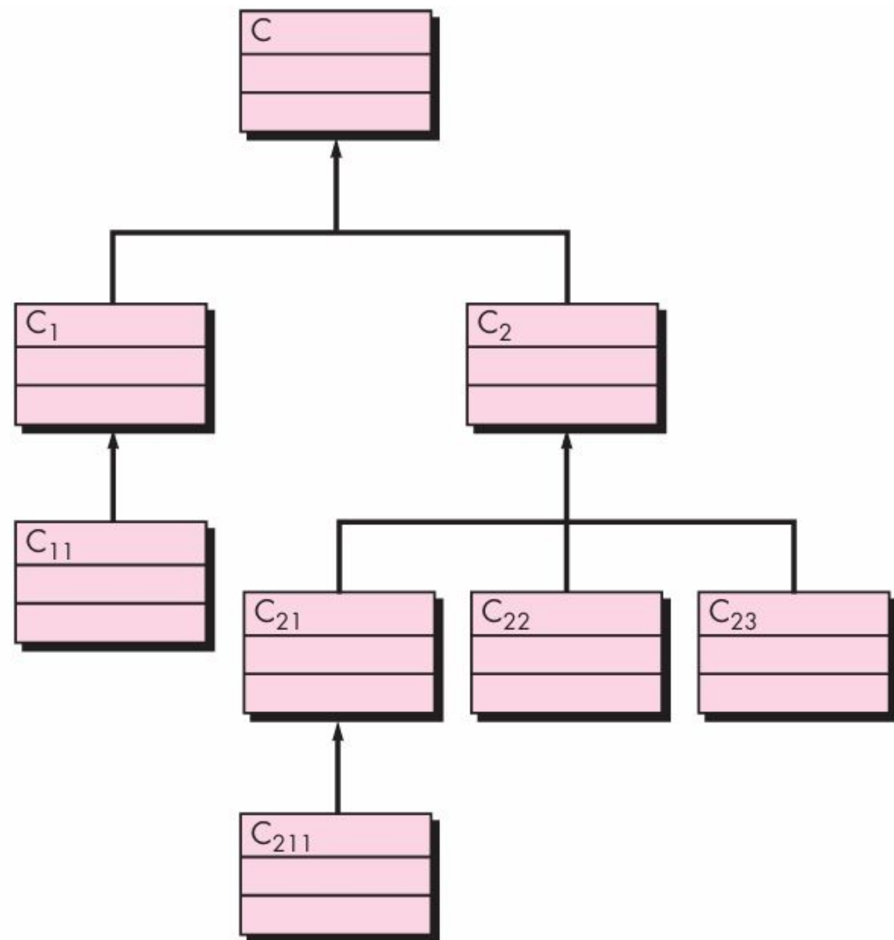


FIGURE 23.5

A class hierarchy

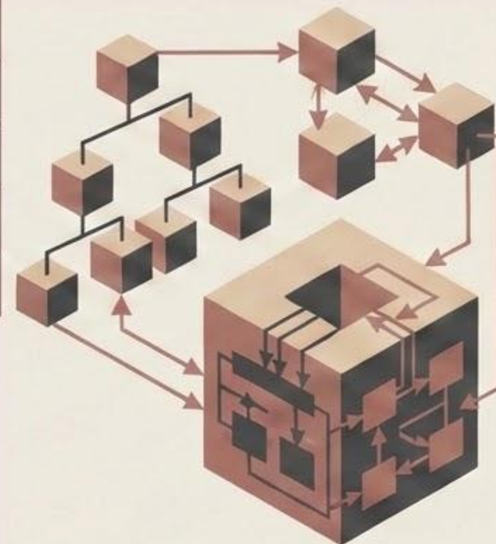


Interpreting OO Metrics

Balancing Trade-offs

Interpreting CK Metrics at a Glance

Metric	Low Value Means...	High Value Means...
WMC	Simple class, easy to maintain.	Complex class, hard to test/maintain.
DIT	Simple, flat hierarchy.	Rich reuse, but behavior is hard to predict.
CBO	Independent, highly reusable.	Tightly coupled, easily broken by outside changes.
LCOM	High cohesion (Doing one thing well).	Low cohesion (A "God class" doing too much).



MOOD Metrics (Metrics for Object-Oriented Design)

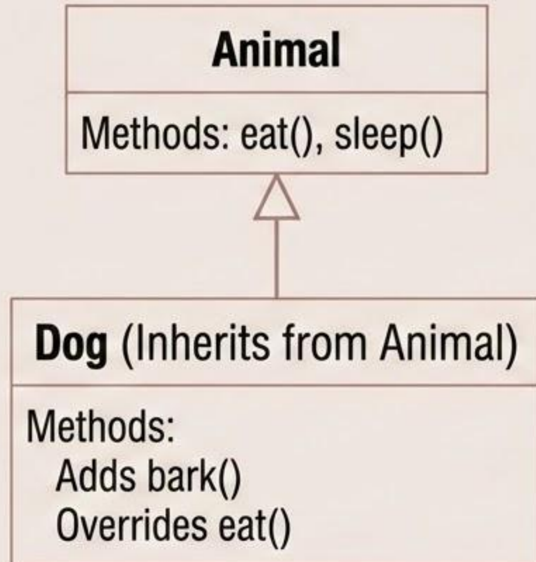
Method Hiding Factor (MHF) / Attribute Hiding Factor (AHF): Proportion of methods/attributes that are properly declared private (Encapsulation).

Method Inheritance Factor (MIF) / Attribute Inheritance Factor (AIF): Proportion of methods/attributes that are inherited rather than newly declared.

In-Class Exercise: Let's Apply the Metrics

Calculating DIT, NOC, and RFC

The Scenario:



Calculate the Following:

DIT for Dog: ?

NOC for Animal: ?

RFC for Dog: ?

Key Takeaways: The Framework

Vocabulary and Alignment

The Hierarchy of Data

Goal-Oriented Measurement (GQM)

Never measure blindly. Always ensure metrics are strictly tied to organizational objectives using the pipeline:

Key Takeaways: Early Estimation

Measuring Before You Code

Function-Based Metrics (Function Points)

- Measures the true size of the software entirely from the user's perspective.
- 100% independent of the programming language.
- Essential for establishing baseline productivity and quality benchmarking.

Specification Quality Metrics

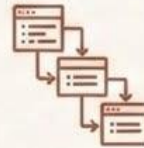
- Tracks defects per page, volatility, and ambiguity.
- Helps assess the actual quality of the requirements before expensive design and coding work begins.

Key Takeaways: Design Metrics

Architecture and Object-Oriented Quality

Architectural Design Metrics

- Evaluates the macro-level system.
- Focuses on minimizing structural complexity and maximizing the rule of Low Coupling + High Cohesion.



Object-Oriented Design Metrics (CK Metrics)

- Evaluates the micro-level class design.
- Uses metrics like WMC, DIT, NOC, CBO, RFC, and LCOM.
- Highly predictive of a system's future maintainability, reusability, and testability.

Conclusion

Steering Toward Quality

“What gets measured gets managed.”

The Alternative

Without metrics, you are navigating the software development lifecycle completely blindly, relying purely on subjective guesswork.

The Advantage

With metrics, you have the objective data required to predict risks, control your processes, and actively steer your team toward total software quality.