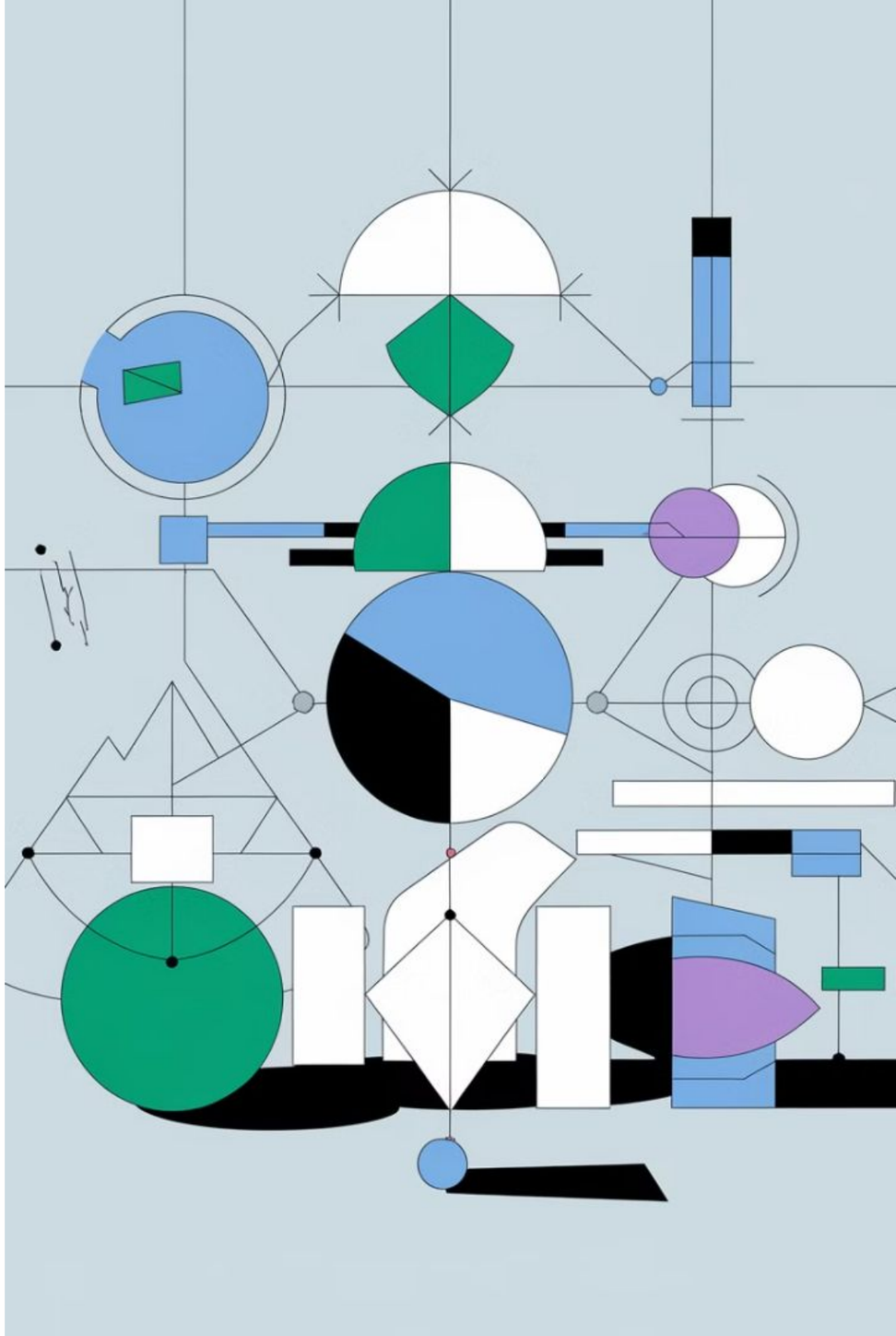
Product Metrics for Software Engineering

Product metrics provide software engineers with quantitative ways to assess the quality of internal product attributes, enabling quality assessment before a product is built. This presentation explores various types of product metrics for requirements models, design, source code, testing, and maintenance. We'll examine how metrics can provide insights to create effective models, solid code, and thorough tests across different types of software systems.



Software Measurement

McCall’s quality factors were proposed in the early 1970s. They are as valid today as they were in that time. It’s likely that software built to conform to these factors will exhibit high quality well into the 21st century, even if there are dramatic changes in technology.

The Challenge of Product Metrics

1

Search for Comprehensive Metrics

Over the past four decades, many researchers have attempted to develop a single metric that provides a comprehensive measure of software complexity. Fenton characterizes this research as a search for "the impossible holy grail."

2

Multiple Perspectives

Dozens of complexity measures have been proposed, each taking a somewhat different view of what complexity is and what attributes of a system lead to complexity.

3

Analogy to Car Evaluation

By analogy, consider a metric for evaluating an attractive car. Some observers might emphasize body design; others might consider mechanical characteristics; still others might tout cost, or performance, or the use of alternative fuels, or the ability to recycle when the car is junked.

4

Difficulty in Single Value Metrics

Since any one of these characteristics may be at odds with others, it is difficult to derive a single value for "attractiveness." The same problem occurs with computer software.

Goal-Oriented Software Measurement

1

Goal/Question/Metric (GQM) Paradigm

The Goal/Question/Metric (GQM) paradigm has been developed by Basili and Weiss as a technique for identifying meaningful metrics for any part of the software process.

2

Establishing Goals

GQM emphasizes the need to establish an explicit measurement goal that is specific to the process activity or product characteristic that is to be assessed.

3

Defining Questions

Define a set of questions that must be answered in order to achieve the goal.

4

Identifying Metrics

Identify well-formulated metrics that help to answer these questions.

Characteristics of Effective Software Metrics

1 **Simple and Computable**

Effective software metrics should be relatively easy to learn and compute, not demanding inordinate effort or time.

2 **Empirically and Intuitively Persuasive**

The metric should satisfy the engineer's intuitive notions about the product attribute under consideration.

3 **Consistent and Objective**

The metric should always yield unambiguous results. An independent third party should be able to derive the same metric value using the same information about the software.

4 **Consistent Units and Dimensions**

The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units.

More Characteristics of Effective Metrics

1 **Programming Language Independent**

Metrics should be based on the requirements model, the design model, or the structure of the program itself. They should not be dependent on the vagaries of programming language syntax or semantics.

2 **Effective Feedback Mechanism**

The metric should provide engineers with information that can lead to a higher-quality end product.

3 **Empirically Validated**

Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions.

4 **Scalability**

A metric should "scale up" to large systems and work in a variety of programming languages and system domains.

Function-Based Metrics

1

External Inputs (EIs)

Each external input originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update internal logical files (ILFs).

2

External Outputs (EOs)

Each external output is derived data within the application that provides information to the user. In this context external output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

3

External Inquiries (EQs)

An external inquiry is defined as an online input that results in the generation of some immediate software response in the form of an online output (often retrieved from an ILF).

4

Internal Logical Files (ILFs)

Each internal logical file is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

Function Point Calculation

Function Point Formula

FP = count total * [0.65 + 0.01 * Σ(Fi)]

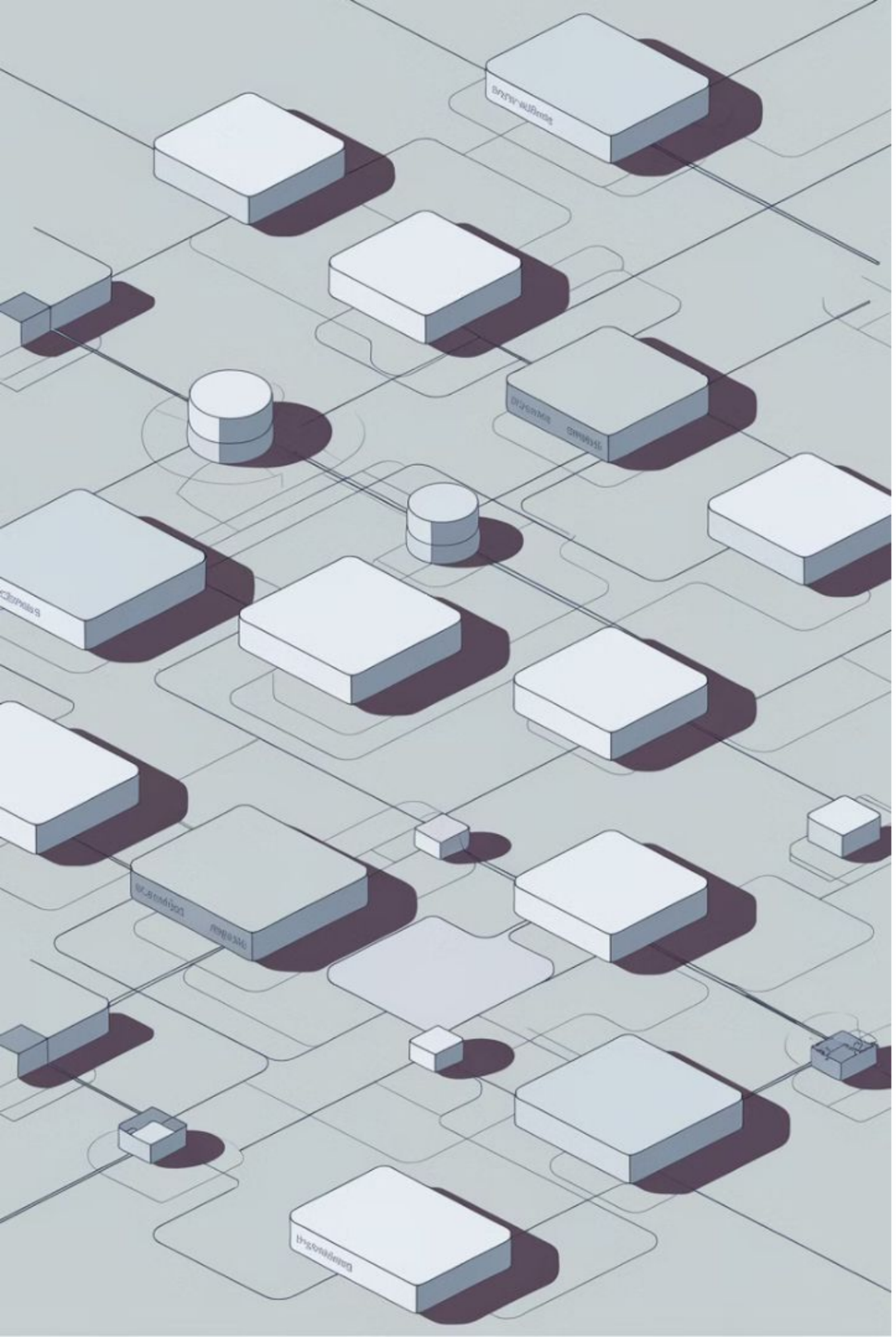
Where count total is the sum of all FP entries obtained from the complexity table, and Fi (i = 1 to 14) are value adjustment factors based on responses to 14 questions about system characteristics.

Value Adjustment Factors

14 questions are answered on a scale from 0 (not important or applicable) to 5 (absolutely essential). These cover aspects like backup and recovery, data communications, distributed processing, performance, and more.

Complexity Table

A table is used to determine the complexity (simple, average, or complex) and corresponding weight for each type of function (EIs, EOs, EQs, ILFs, EIFs). The weighted counts are summed to get the count total.



Object-Oriented Design Metrics: Characteristics

Size

number of population (classes, methods, attributes), volume (active objects, method invocations, messages exchanged), length (inheritance tree, dependencies), and functionality (use cases, features, functional requirements)

Coupling

Represents the physical connections between elements of the OO design, such as the number of collaborations between classes or messages passed between objects.

Sufficiency

The degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application.

More Object-Oriented Design Metrics Characteristics

Completeness

Similar to sufficiency, but considers multiple points of view. It asks: "What properties are required to fully represent the problem domain object?" This has implications for reusability.

Cohesion

Determined by how well the operations (methods) and properties (attributes) of a class work together to achieve a single, well-defined purpose.

Primitiveness

The degree to which an operation is atomic - that is, it cannot be constructed out of a sequence of other operations contained within a class. A highly primitive class encapsulates only primitive operations.

Similarity

The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose.

CK Metrics Suite: Weighted Methods per Class (WMC)

Definition

WMC = Σci for i = 1 to n, where n is the number of methods and ci is the complexity of each method. The complexity metric should be normalized so that nominal complexity for a method takes on a value of 1.0.

Interpretation

WMC is an indicator of the amount of effort required to implement and test a class. Larger WMC values suggest more complex inheritance trees and potentially less reusable classes.

Considerations

Developing a consistent counting approach for methods is important. WMC should be kept as low as reasonable while still fulfilling the class's responsibilities.

CK Metrics Suite: Depth of Inheritance Tree (DIT)

1 **Definition**

DIT is the maximum length from a node to the root of the inheritance tree.

2 **Implications**

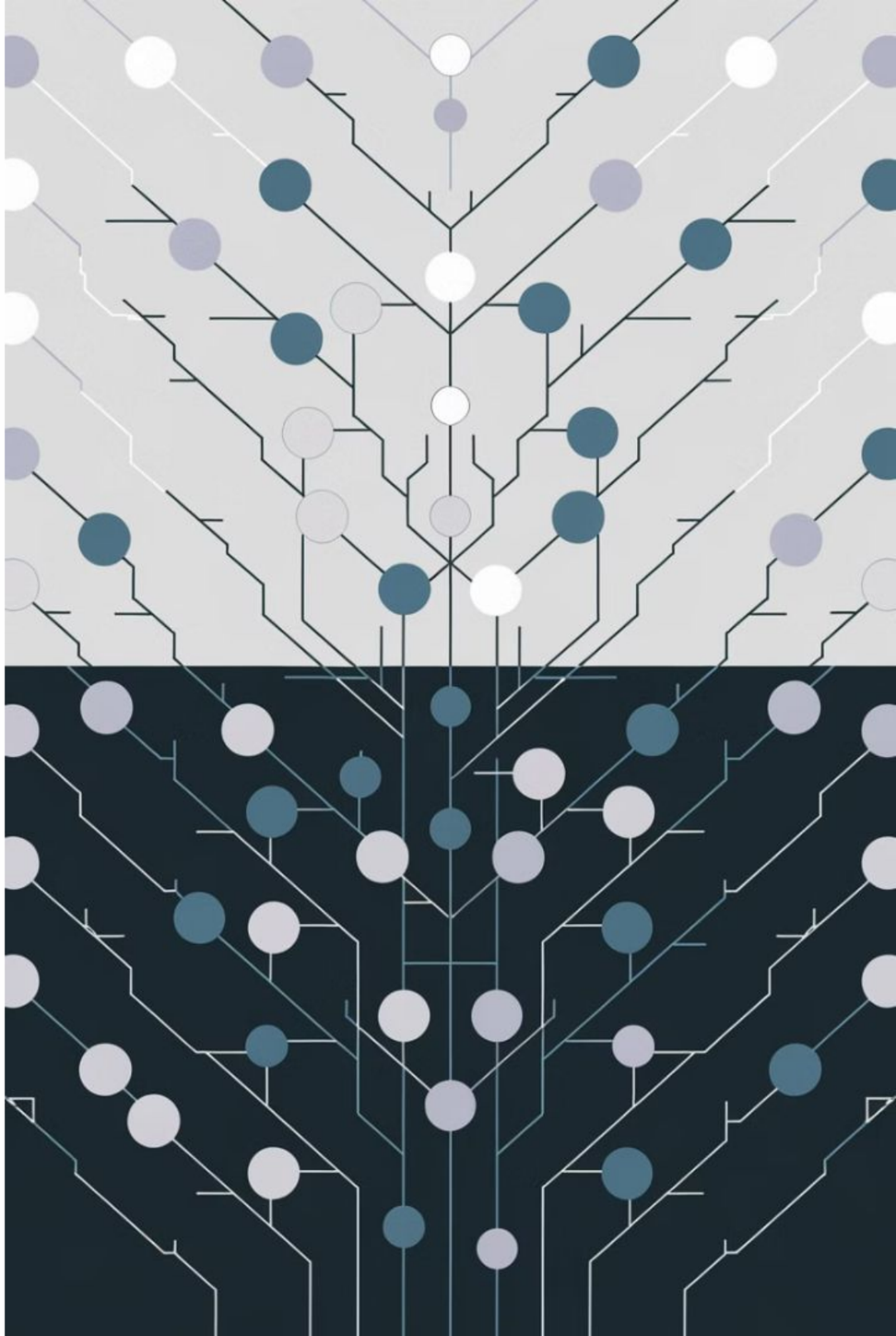
As DIT grows, lower-level classes are likely to inherit many methods, potentially making it difficult to predict class behavior.

3 **Design Complexity**

A deep class hierarchy (large DIT) leads to greater design complexity.

4 **Reusability**

Large DIT values imply that many methods may be reused through inheritance.



CK Metrics Suite: Number of Children (NOC)

Definition

NOC is the number of immediate subclasses subordinate to a class in the class hierarchy.

Reuse Implications

As NOC increases, the potential for reuse through inheritance also increases.

Abstraction Concerns

High NOC values may dilute the abstraction of the parent class if some children are not appropriate members.

Testing Impact

As NOC increases, the amount of testing required to exercise each child in its operational context will also increase.

CK Metrics Suite: Coupling Between Object Classes (CBO)

Definition

CBO is the number of collaborations listed for a class on its CRC index card.

Reusability Impact

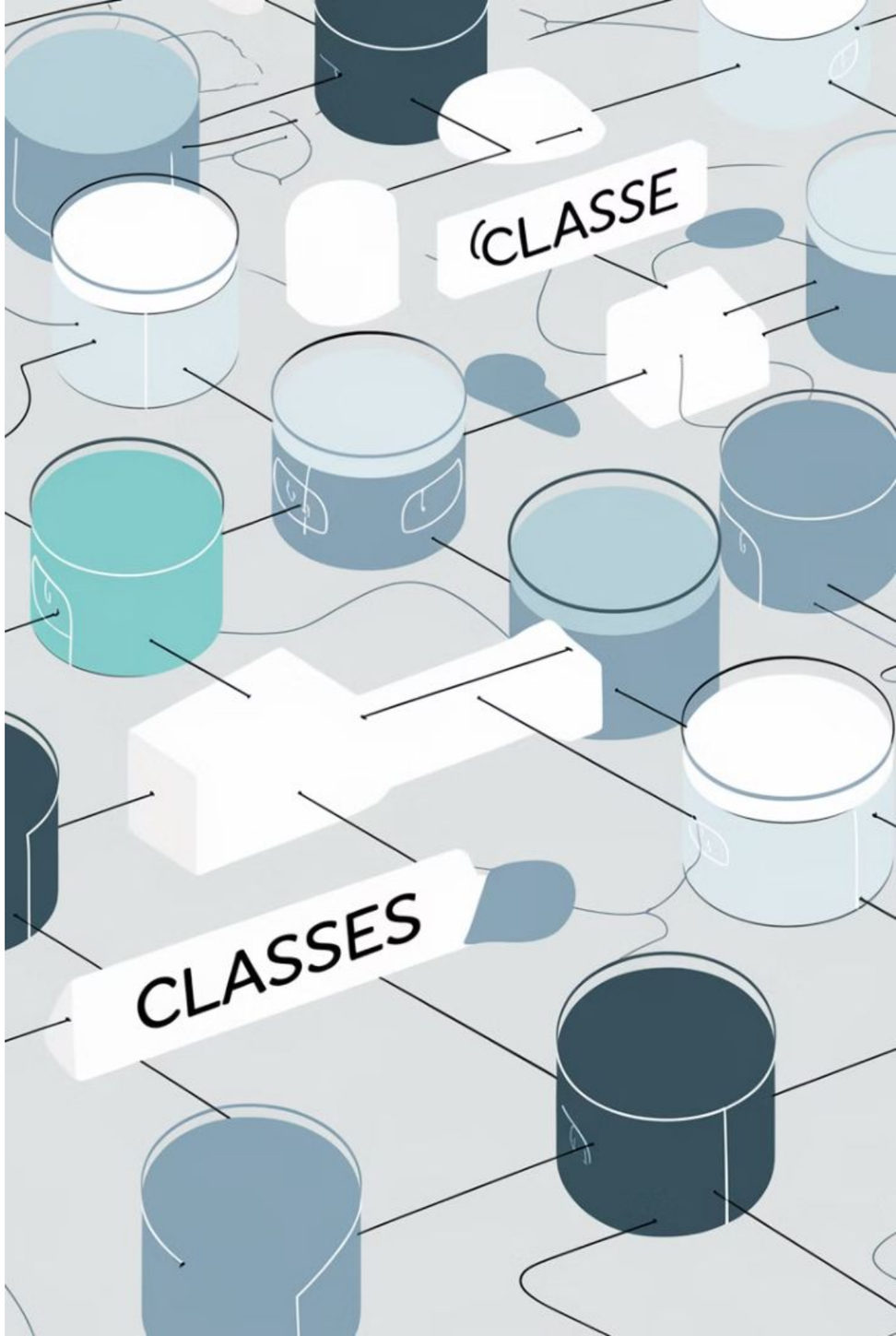
As CBO increases, it is likely that the reusability of a class will decrease.

Modification Complexity

High values of CBO complicate modifications and the testing that ensues when modifications are made.

Design Guideline

CBO values for each class should be kept as low as is reasonable. This is consistent with the general guideline to reduce coupling in conventional software.



CK Metrics Suite: Response For a Class (RFC)

1 Definition

RFC is the number of methods in the response set of a class. The response set is "a set of methods that can potentially be executed in response to a message received by an object of that class".

2 Testing Effort

As RFC increases, the effort required for testing also increases because the test sequence grows.

3 Design Complexity

As RFC increases, the overall design complexity of the class increases.

4 Comprehension Difficulty

Classes with high RFC values may be more difficult to understand and maintain.

CK Metrics Suite: Lack of Cohesion in Methods (LCOM)

Definition

LCOM is the number of methods that access one or more of the same attributes. If no methods access the same attributes, then LCOM = 0.

Interpretation

High LCOM values indicate that methods may be coupled to one another via attributes, increasing the complexity of the class design.

Design Goal

It is generally desirable to keep cohesion high, which means keeping LCOM low.

Considerations

While low LCOM is generally preferred, there are cases where high LCOM can be justified based on the class's responsibilities.

MOOD Metrics Suite: Method Inheritance Factor (MIF)

1

Definition

MIF = Σ Mi(Ci) / Σ Ma(Ci), where Mi(Ci) is the number of methods inherited in class Ci, and Ma(Ci) is the total number of methods that can be invoked in association with Ci

2

Interpretation

Higher MIF values suggest greater complexity and greater testing efforts.

MOOD Metrics Suite: Coupling Factor (CF)

Definition

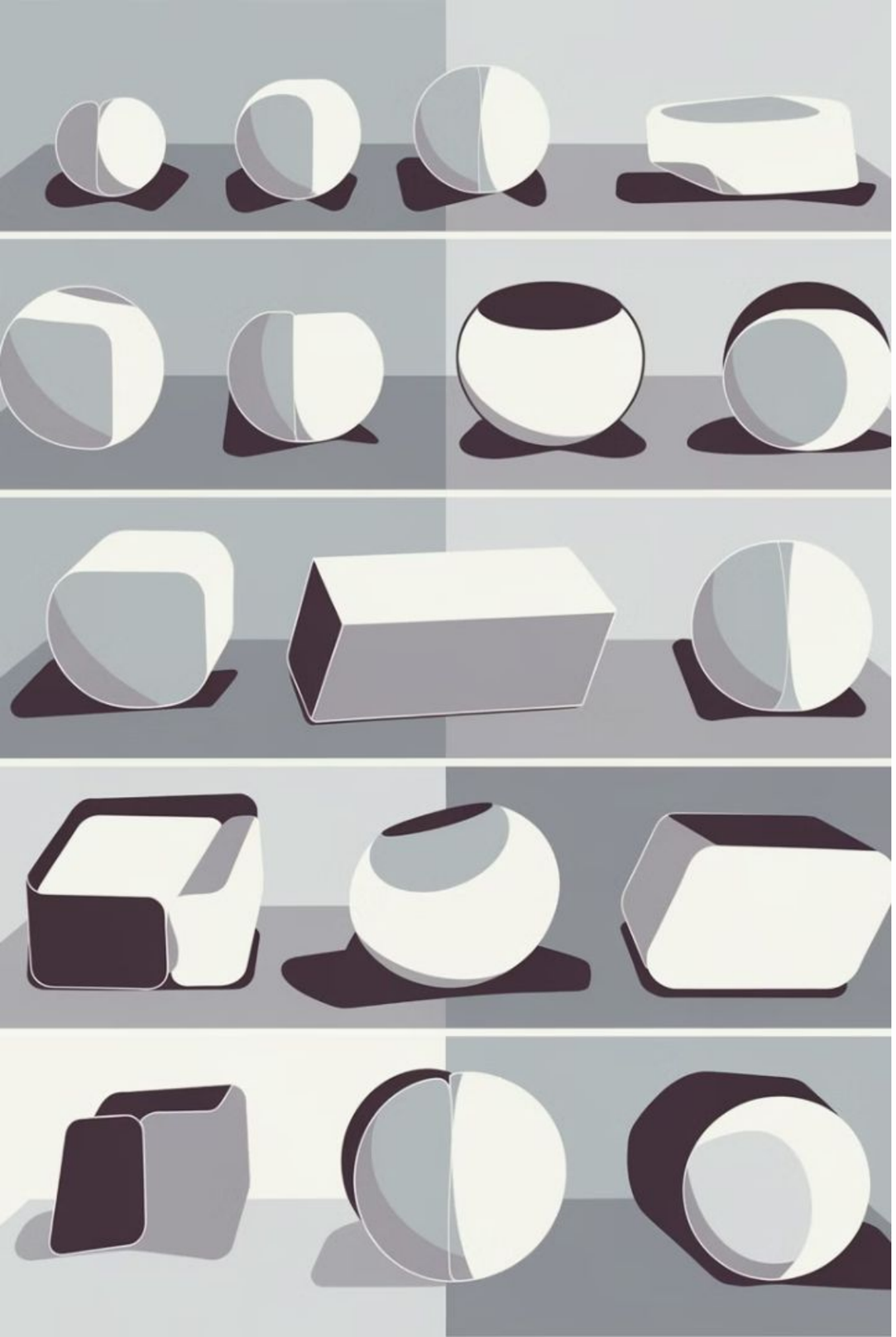
CF = Σi Σj is_client(Ci, Cj) / (TC^2 - TC), where is_client(Ci, Cj) = 1 if there's a relationship between client class Ci and server class Cj, and 0 otherwise. TC is the total number of classes.

Interpretation

As CF increases, the complexity of the OO software will also increase, potentially reducing understandability, maintainability, and reusability.

Design Implications

Lower CF values are generally desirable, indicating a more modular and loosely coupled design.



Lorenz and Kidd Metrics: Class Size (CS)

Definition

CS can be measured by: 1) The total number of operations (both inherited and private instance operations) encapsulated within the class, and 2) The number of attributes (both inherited and private instance attributes) encapsulated by the class.

Interpretation

Large values for CS indicate that a class may have too much responsibility, which can reduce reusability and complicate implementation and testing.

Weighting

Inherited or public operations and attributes should be weighted more heavily in determining class size.

Averages

Averages for the number of class attributes and operations may be computed. Lower average values for size generally indicate higher potential for reuse.

User Interface Design Metrics

Layout Appropriateness (LA)

- Evaluates how effectively the graphical user interface (GUI) layout supports user interactions. It does this by considering the absolute and relative positioning of interface elements, their frequency of use, and the cost of transitions between these elements..
- Simple characteristics like the number of words, links, graphics, colors, and fonts can affect the perceived complexity and quality of a web page.
- Nielsen and Levy report a high correlation between users' average task performance and their subjective satisfaction with a GUI.

Layout Appropriateness (LA)

Absolute Position of Layout Entities:

1. Refers to the fixed location of an element on the screen (e.g., top-left corner, center).
2. It affects the user's ability to find and access an element quickly.

Relative Position of Layout Entities:

3. Examines the spatial relationship between elements.
4. Related elements (e.g., "Save" and “Save as" buttons) should be positioned near each other.

Frequency of Use:

1. Elements used more frequently should be more prominent and easier to access.
2. For example, a "Search" bar in an e-commerce site should be centrally located and easily visible.

Cost of Transitions Between Entities:

3. Measures the effort required to move from one element to another.
4. Includes physical distance on the screen, as well as cognitive load (how intuitive the transition feels).

Halstead's Software Science Metrics

Basic Measures

n1 = number of distinct operators, n2 = number of distinct operands, N1 = total operator occurrences, N2 = total operand occurrences

Derived Metrics

Program length: N = n1 log2 n1 + n2 log2 n2 Program volume: V = N log2 (n1 + n2) Volume ratio: L = (2/n1) * (n2/N2)

Applications

These metrics can be used to estimate program length, minimum volume, actual volume, program level, language level, development effort, and even projected number of faults.

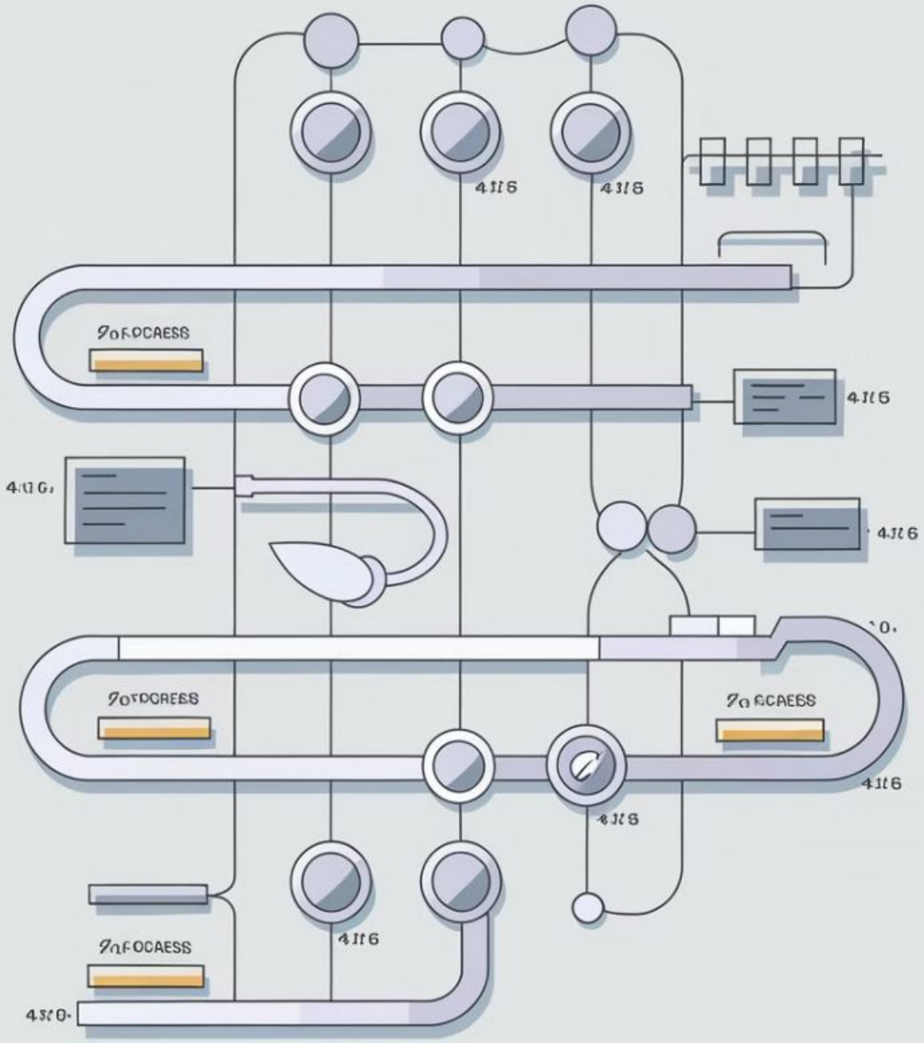
Metrics for Testing

Cyclomatic Complexity (CC)

Cyclomatic complexity is a quantitative metric that measures the complexity of a program's control flow. Modules with high cyclomatic complexity are more likely to be error-prone. And more testing efforts will be needed

$$CC = E - N + 2$$

- E: Number of edges in the control flow graph.
- N: Number of nodes in the control flow graph.

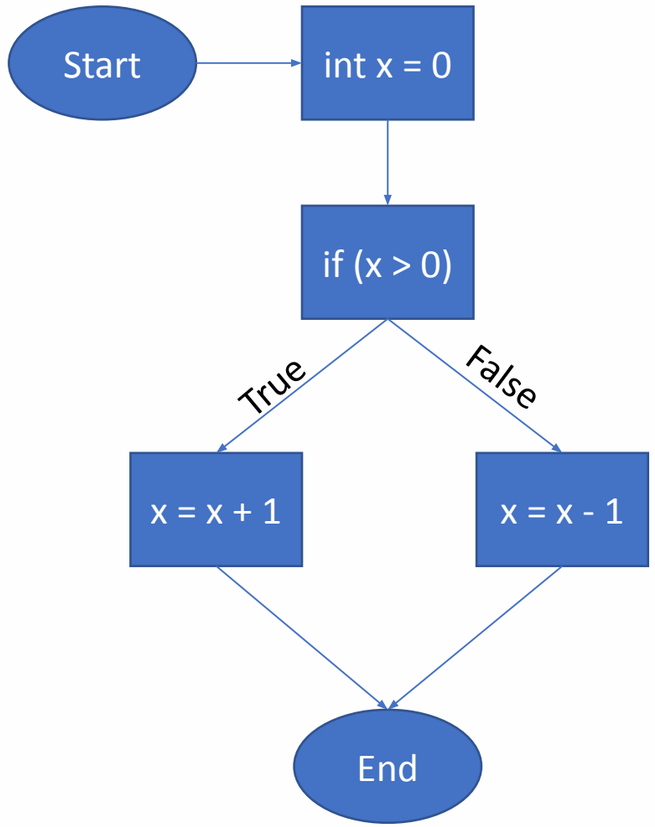


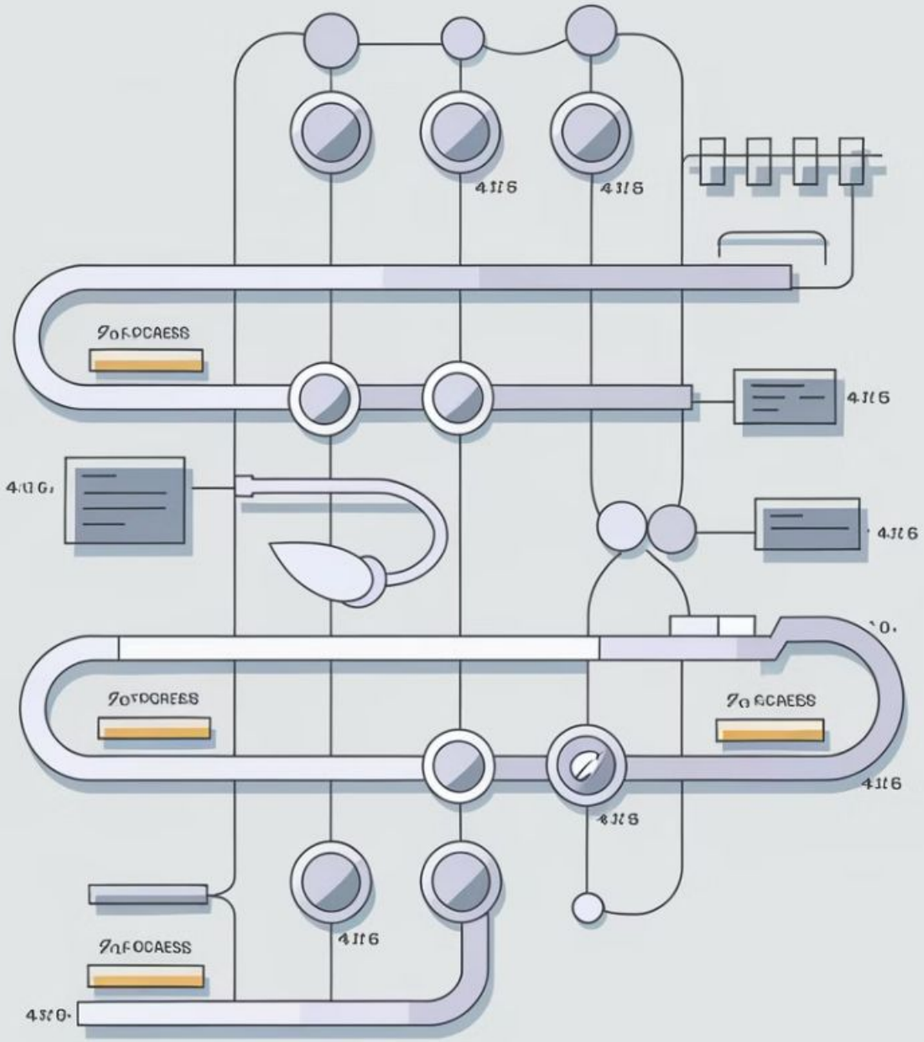
Metrics for Testing

Cyclomatic Complexity (CC)

```
int x = 0;
if (x > 0) {
   x = x + 1;
} else {
   x = x - 1;
}
```

$$CC = E - N + 2$$
$$= 6 - 6 + 2$$
$$= 2$$





OO Testing Metrics

- LCOM (Lack of Cohesion in Methods)

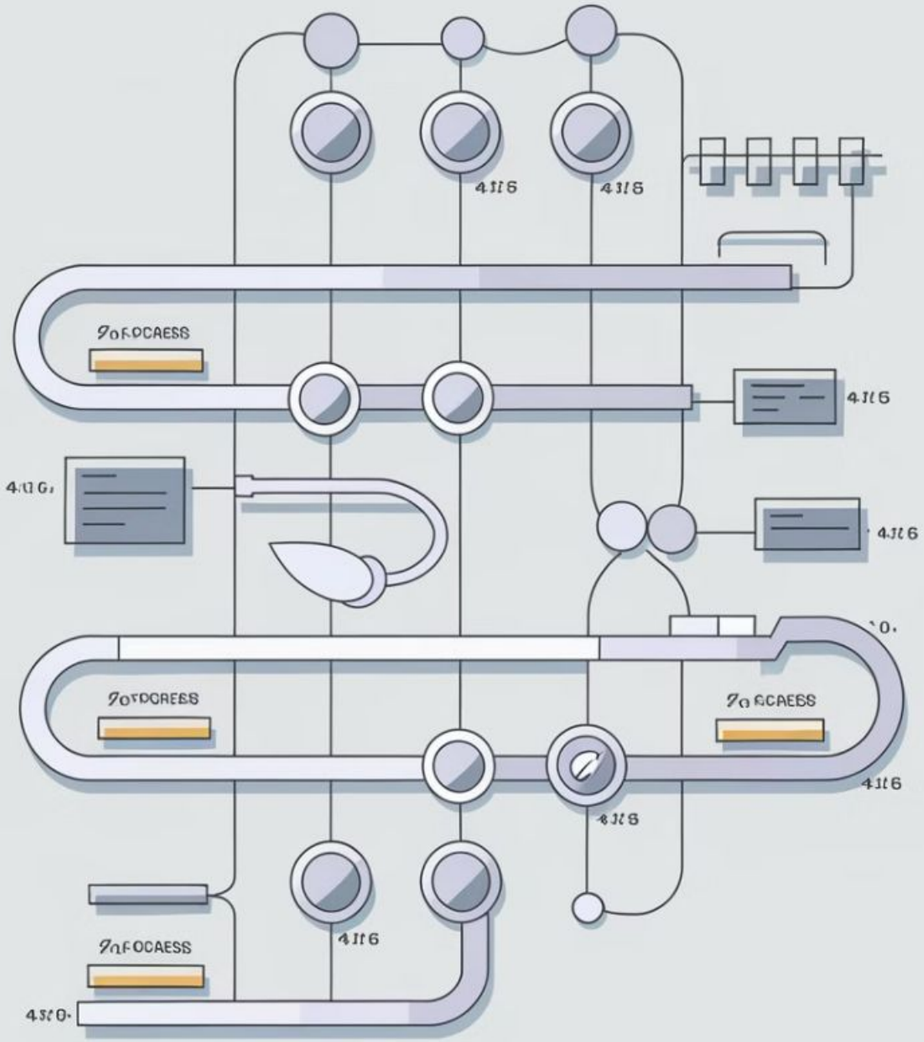
Definition: Measures the degree of relatedness between methods in a class.
Reason: Higher LCOM indicates more states, increasing the likelihood of side effects.
Impact: More extensive testing is needed to ensure side effects are avoided.

- PAP (Percent Public and Protected)

Definition: Indicates the percentage of public and protected attributes in a class.
Reason: High PAP increases coupling, leading to greater risk of side effects.
Impact: Testing must uncover unintended interactions among classes.

- PAD (Public Access to Data Members)

Definition: Measures the number of external classes accessing a class's attributes.
Reason: High PAD violates encapsulation, increasing the risk of side effects.
Impact: Tests must address potential cross-class side effects.



OO Testing Metrics

- NOR (Number of Root Classes)

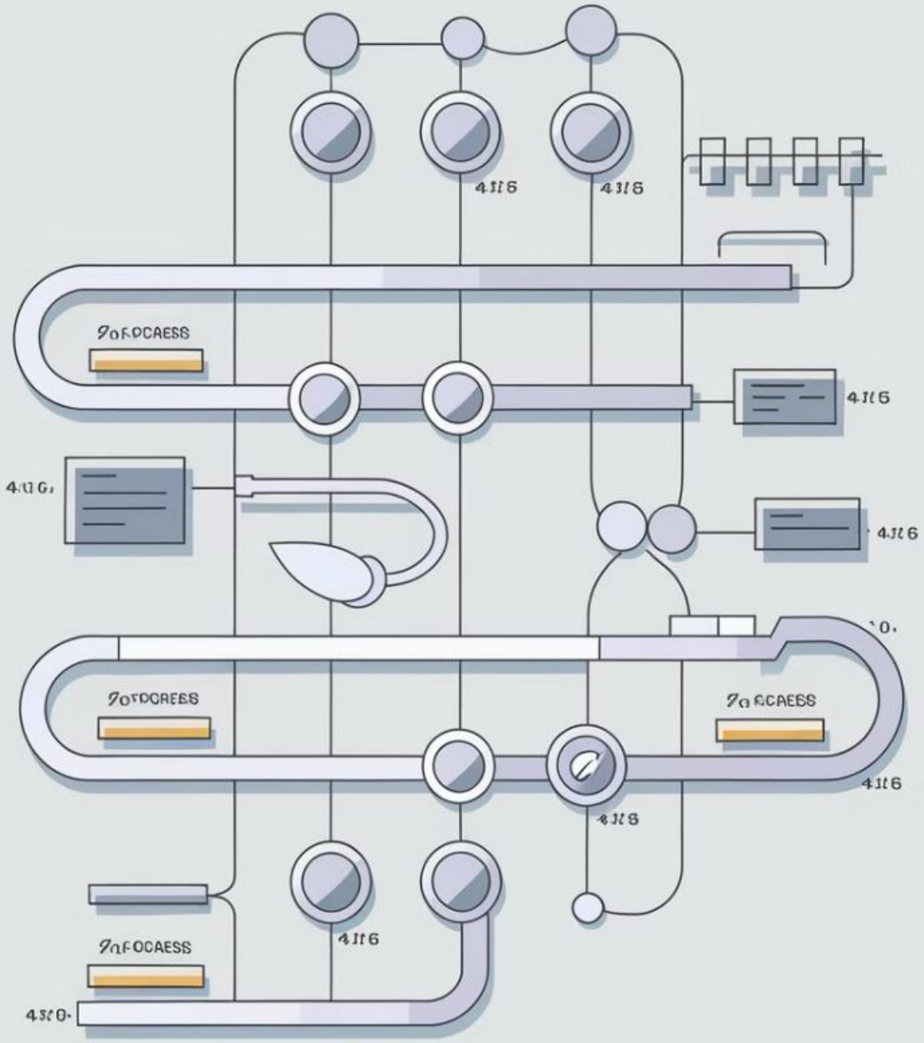
Definition: Counts the distinct class hierarchies in the design model.
Reason: As NOR increases, the complexity of testing each hierarchy grows.
Impact: More root classes mean more effort in creating comprehensive test suites.

- FIN (Fan-In)

Definition: Measures the number of parent classes a class inherits from.
Reason: FIN > 1 (Multiple inheritance) increases complexity and risks ambiguity.
Impact: Should be avoided; simplifies testing and design.

- NOC & DIT (Number of Children & Depth of Inheritance Tree)

Definition: NOC: Number of immediate subclasses; DIT: Levels in inheritance tree.
Reason: Deep inheritance and numerous subclasses require retesting of superclass methods.
Impact: Increases testing workload for each additional layer or subclass.



Metrics for Software Quality

The primary goal of software engineering is to produce high-quality systems, applications, or products that meet market needs within a specified timeframe. This requires applying effective methods, modern tools, and mature software processes. Measurement is crucial for assessing and achieving high quality across all aspects of software development, from requirements to testing.

Measuring Quality

Correctness

- Degree to which the software performs its required function.
- The most common measure for correctness is defects per KLOC
- Defect Density per KLOC = number of verified defects (bugs or errors) in the software per 1,000 lines of code

Maintainability

- the ease with which a program can be corrected, adapted, or enhanced
- A simple time-oriented metric is mean-time-to-change (MTTC)
- MTTC = average time required to implement a change

Integrity

- system’s ability to withstand attacks to its security
- Integrity=1−(threat×(1−security))
- Threat -> probability that a specific type of attack will occur, Security -> probability an attack will be repelled

Usability

- ease of use with which users can achieve their goals

Defect Removal Efficiency (DRE)

DRE is a quality metric that measures the effectiveness of quality assurance in identifying and removing defects during the software development lifecycle.

$$\text{DRE} = \frac{E+D}{E}$$

E: Errors found before delivery (via reviews, testing, etc.)

D: Defects found after delivery (by the end user).

Focus on achieving DRE values close to 1 in all phases to minimize defects and improve overall software quality.

Reference

- Chapter 23 – Product Metrics
- Chapter 25.3 – Metrics for Software Quality

  Software Engineering A Practitioner’s Approach, R. Pressman

THANK
YOU

Software Project Estimation

Software project estimation is a critical part of project planning.

Before the project can begin, the software team should estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish.

Project Planning Task Set-I

- Establish project scope
- Determine feasibility
- Analyze risks
- Define required resources
  - Determine require human resources
  - Define reusable software resources
  - Identify environmental resources

Project Planning Task Set-II

- Estimate cost and effort
  - Decompose the problem
  - Develop two or more estimates using size, function points, process tasks or use-cases
  - Reconcile the estimates
- Develop a project schedule
  - Scheduling is considered in detail in Chapter 27.
    - Establish a meaningful task set
    - Define a task network
    - Use scheduling tools to develop a timeline chart
    - Define schedule tracking mechanisms

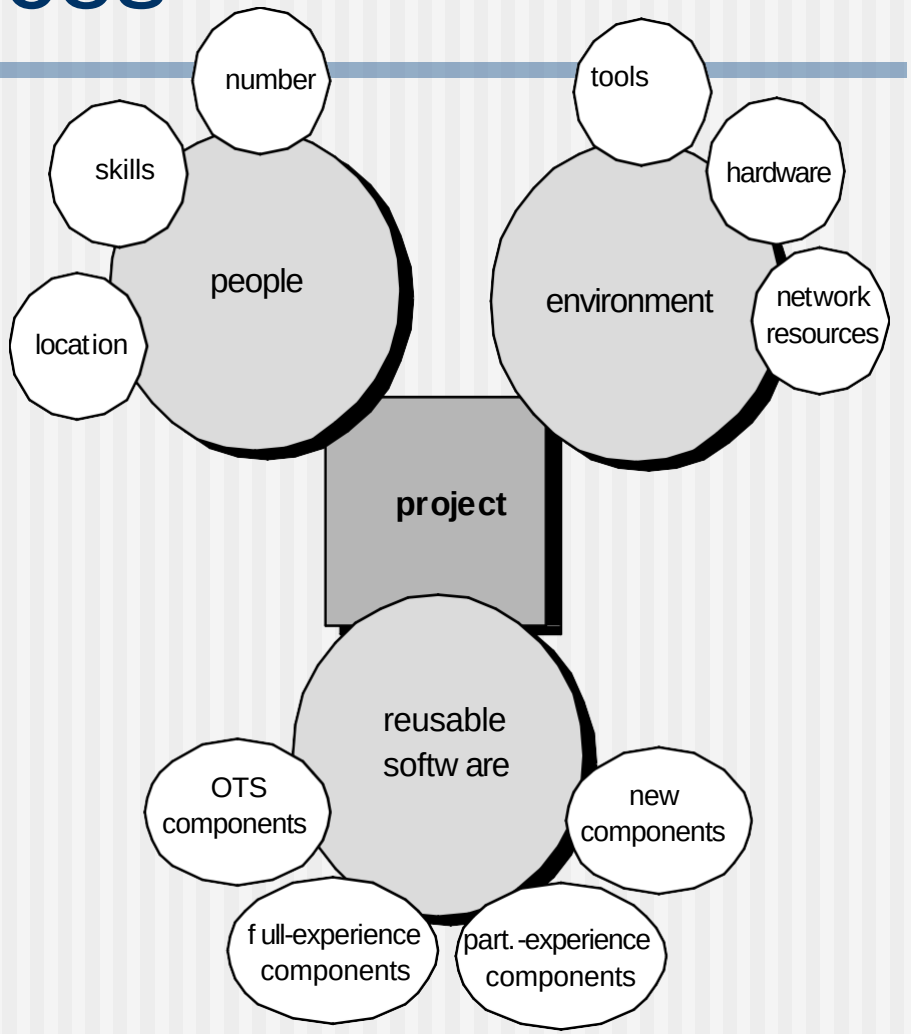
What is Scope?

- *Software scope* describes
  - the functions and features that are to be delivered to end-users
  - the data that are input and output
  - the “content” that is presented to users as a consequence of using the software
  - the performance, constraints, interfaces, and reliability that *bound* the system.
- Scope is defined using one of two techniques:
  - A narrative description of software scope is developed after communication with all stakeholders.
  - A set of use-cases is developed by end-users.

To Understand Scope ...

- Understand the customers needs
- understand the business context
- understand the project boundaries
- understand the customer’s motivation
- understand the likely paths for change

Resources


number
skills
people
location
tools
hardware
environment
network resources
project
reusable softw are
OTS components
new components
f ull-experience components
part. -experience components

Estimation Techniques

- Past (similar) project experience
- Conventional estimation techniques
  - task breakdown
  - size (e.g., LOC, FP) estimates
- Empirical models

Conventional Methods: LOC/FP Approach

- compute LOC/FP using estimates of information domain values
- use historical data to build estimates for the project

Example: LOC Approach

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

If Average productivity for the system = 620 LOC/pm and labor rate =$8000 /pm:

Cost per LOC = $8,000 / 620 = $13

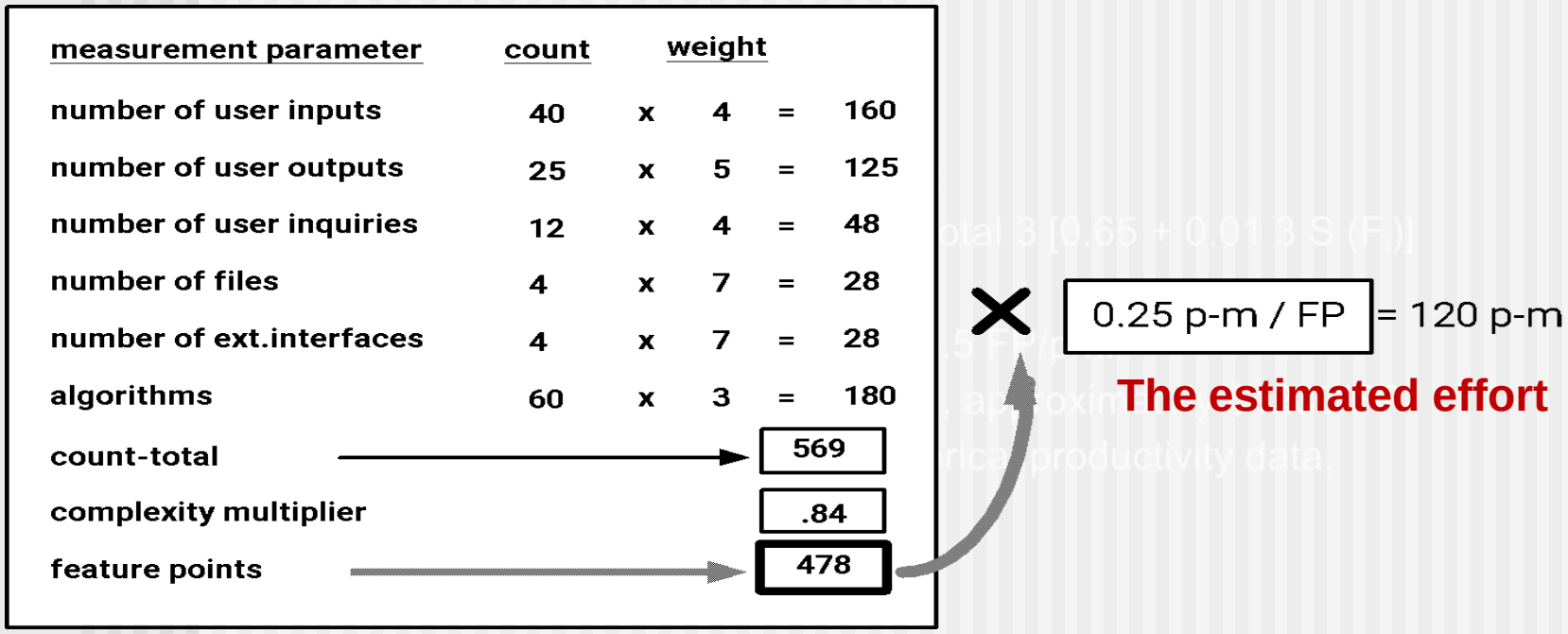
Total estimated project cost = 33,200 x $13 = $431,000

The estimated effort = 33,200 / 620

= 54 person-months.

Example: FP Approach

Effort = Feature Points × Productivity Rate



Empirical Estimation Models

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.

COCOMO-II

- COCOMO-II (***CO****nstructive* ***CO****st* ***MO****del*) is an **empirical estimation model** used in software engineering to predict the effort, cost, and schedule required for a software project.
- It offers three sub-models, each tailored to different stages of software development:
  - *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
  - *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
  - *Post-architecture-stage model.* Used during the construction of the software.

COCOMO-II

Application Composition Model

Effort (Person-Months) =(Object Points) / Productivity Rate

Object Points: A count of the number and complexity of screens or reports.
Productivity Rate: Adjusted based on team experience and tool efficiency.

COCOMO-II

Application Composition Model

Developing an employee management system with:

- **20 screens** (medium complexity, weight = 2)
- **5 reports** (high complexity, weight = 3)

Step 1: Calculate Object Points:

Object Points=(20×2)+(5×3)=40+15=55

Step 2: Select Productivity Rate:

Productivity rate = 7 (Assuming moderate skill levels and standard tools)

Step 3: Calculate Effort:

Effort (Person-Months)==55/7≈7.86 person-months.

COCOMO-II

Application Composition Model

Component Type	Weight for Low Complexity	Weight for Medium Complexity	Weight for High Complexity
Screens	1	2	3
Reports	2	3	4

Factor	Range (Object Points/Person-Month)	Description
High Productivity	12–15	Skilled team with advanced tools and efficient processes.
Medium Productivity	7–12	Average team capability and standard tool support.
Low Productivity	3–7	Inexperienced team or limited tools and process support.

COCOMO-II

Early Design Model & Post-Architecture Model

$$\text{Effort} = A \times \text{Size}^{B} \times \prod(\text{EM})$$

Size: Measured LOC
A: Tuning coefficient
B: Scale factor
∏(EM) : Product of effort multipliers for the project.

COCOMO-II

Inputs

- **Size of the Project (LOC)**:
  1. Estimated size is **25,000 lines of code (LOC)**.
- **Tuning Coefficient (A)**:
  1. A=2.8 (Based on the project complexity and domain)
- **Scale Factor (B)**:
  1. B=1.12 ((medium-sized project with moderate complexity)
- **Effort Multipliers (∏EM)**:
  1. **Required Software Reliability (RELY)**: High reliability is needed (1.10).
  2. **Platform Volatility (PVOL)**: Low platform changes expected (0.87).
  3. **Analyst Capability (ACAP)**: Highly skilled analysts (0.85).
  4. **Use of Modern Tools (TOOL)**: Moderately advanced tools (0.90).
  5. **Required Development Schedule (SCED)**: Nominal (1.00).

∏EM = 1.10×0.87×0.85×0.90×1.00=0.817

Effort (Person-Months)=*A*×Size^*B*×∏*EM* = 2.8×(25)^1.12×0.817 = 70.89

The Software Equation

A dynamic multivariable model

$$E = [LOC \times B^{0.333}/P]^3 \times (1/t^4)$$

where

E = effort in person-months

t = project duration in months

B = “special skills factor”

P = “productivity parameter”

The Software Equation

B (special skills factor)

- *B* increases slowly as "the need for integration, testing, quality assurance, documentation, and management skills grows" [Put92]. For small programs (KLOC 5 to 15), *B* 0.16. For programs greater than 70 KLOC, *B* 0.39.

*P (*productivity parameter)

- Reflects process maturity, engineering practices, programming language level, software environment, team expertise, and application complexity.
- Typical P values: 2000 (real-time embedded software), 10000 (telecom systems), 28000 (business applications).

The Software Equation

Simplified Version

$$t_{\min} = 8.14 \frac{\text{LOC}}{p^{0.43}} \text{ in months for } t_{\min} > 6 \text{ months}$$

$$E = 180\, Bt^3 \text{ in person-months for } E \geq 20 \text{ person-months}$$

Example

$$t_{\min} = 8.14 \times \frac{33{,}200}{12{,}000^{0.43}} = 12.6 \text{ calendar months}$$

$$E = 180 \times 0.28 \times (1.05)^3 = 58 \text{ person-months}$$

The Make-Buy Decision

Acquisition Options

- **Off-the-Shelf Software:**
  Pre-built software purchased or licensed for immediate use.
- **Modified Components:**
  "Full-experience" or "Partial-experience" components are acquired, then customized and integrated.
- **Custom Software:**
  Built by an external contractor to meet specific requirements.

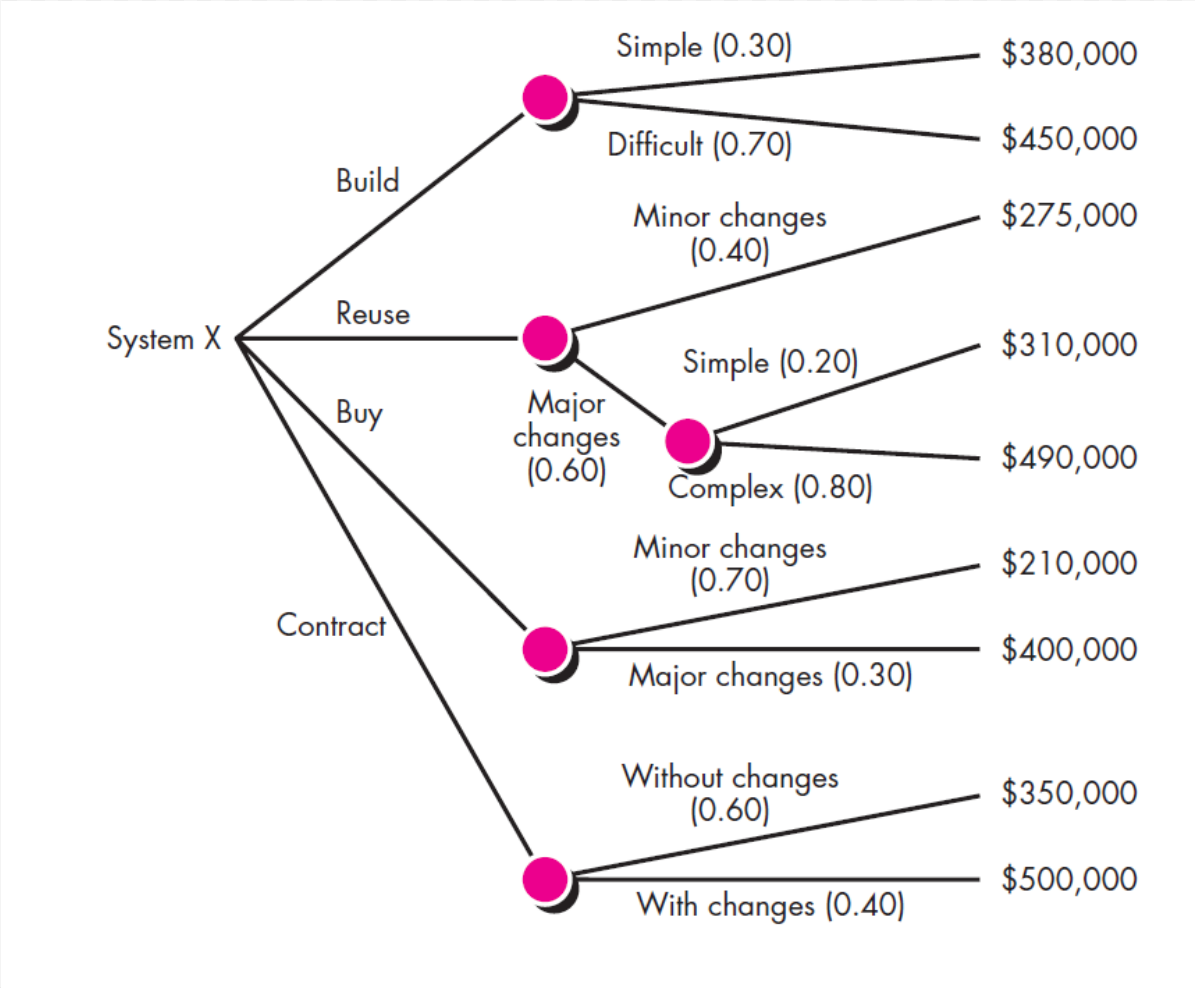
The Make-Buy Decision

Key Factors for Make/Buy Decision

- **Delivery Timeline:**
  Will purchased software be available sooner than internally developed software?
- **Cost Effectiveness:**
  Is the cost of acquisition + customization less than internal development costs?
- **Support Costs:**
  Will external maintenance contracts be cheaper than internal support?

The Make-Buy Decision

- **Decision Tree**



The Make-Buy Decision

- **Expected Cost Estimation:**

$$\text{Expected cost} = \Sigma\,(\text{path probability})_i \times (\text{estimated path cost})_i$$

$$\text{Expected cost}_{\text{build}} = 0.30\ (\$380\text{K}) + 0.70\ (\$450\text{K}) = \$429\text{K}$$

$$\text{Expected cost}_{\text{reuse}} = 0.40\ (\$275\text{K}) + 0.60\ [0.20\ (\$310\text{K}) + 0.80\ (\$490\text{K})] = \$382\text{K}$$

$$\text{Expected cost}_{\text{buy}} = 0.70\ (\$210\text{K}) + 0.30\ (\$400\text{K}) = \$267\text{K}$$

$$\text{Expected cost}_{\text{contract}} = 0.60\ (\$350\text{K}) + 0.40\ (\$500\text{K}) = \$410\text{K}$$

- The lowest expected cost is the “buy” option.

The Make-Buy Decision

Additional Consideration Other than the Cost

- **Availability:**
  - Is the software or component readily available to meet project needs?
- **Experience of Vendor/Developer:**
  - Does the provider have the expertise and a reliable track record?
- **Requirement Conformance:**
  - Does the software align with the functional and non-functional requirements?
- **Local Considerations ("Politics"):**
  - Are there organizational or regional preferences impacting the decision?
- **Likelihood of Change:**
  - How adaptable is the solution to future changes in requirements or technology?

Reference

- Chapter 26 **- Estimation for Software Projects**

  Software Engineering: A Practitioner's Approach, 7/e

  by Roger S. Pressman