SE345

Atılım University
Dept of Software Engineering

Asst. Prof. Dr. Aylin AKCA-OKAN

Tentative Course Schedule

Wk	Subjects	Chapter
1.	Introduction to Software Quality and Assurance	Chapter 1
2.	Introduction to Software Quality and Assurance	Chapter 1
3.	Software Quality Factors	Chapter 3
4.	Overview of Components of the SQA System	Chapter 4
5.	Overview of Components of the SQA System	Chapter 4
6.	Integrating Quality Activities in Project Life Cycle	Chapter 7
7.	Midterm	
8.	Reviews, Inspection and Audits	Chapter 8
9.	Software Quality Metrics	Chapter 21
10.	Procedures and Work Instructions	Chapter 14
11.	Presentations	
12.	Software Change Process	Chapter 18
13.	SOA Process Standards	Chapter 23

(Sommerville; http://www.utdallas.edu/~chung/SE3354Honors/IEEInaugural.pdf)

Measurement of products or systems is absolutely fundamental to the engineering process.

I am convinced that measurement as practised in other engineering disciplines is *IMPOSSIBLE* for software engineering

So, what do you think?

... collecting metrics is too hard

... it's too time-consuming

... it's too political

... it won't prove anything

.....

- to characterise
- to evaluate
- to predict
- to improve

What to measure

- Process
  Measure the efficacy of processes. What works, what doesn't.
- Project
  Assess the status of projects. Track risk. Identify problem areas. Adjust workflow.
- Product
  Measure predefined product attributes (generally related to ISO9126 Software Characteristics)

Measurement, Measures, Metrics

- **Measurement** = how you obtained the number
- **Measure** = raw number
- **Metric** = calculation using the number
- **Indicator** = metric that *signals* performance

Measurement

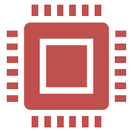
is the act of obtaining a measure

Measure

provides a quantitative indication of the size of some product or process attribute, E.g., Number of errors

A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process

An example measure might be five centimetres.



Metric

is a quantitative measure of the degree to which a system, component, or process possesses a given attribute *(IEEE Software Engineering Standards 1993) : Software Quality* - E.g., Number of errors found per person hours expended
An example of a metric would be that there were only two user-discovered errors in the first 18 months of operation.

Indicator

is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself

In software terms, an indicator may be a substantial increase in the number of defects found in the most recent release of code.

Can you quantify security, evolvability, ...?

Theory of Measurement

Measurement is a mapping from the empirical, observable world to the formal, relational world.

static measures

- derived from examination of the software itself, e.g., source code.

dynamic measures

- derived from observations of the execution of the software

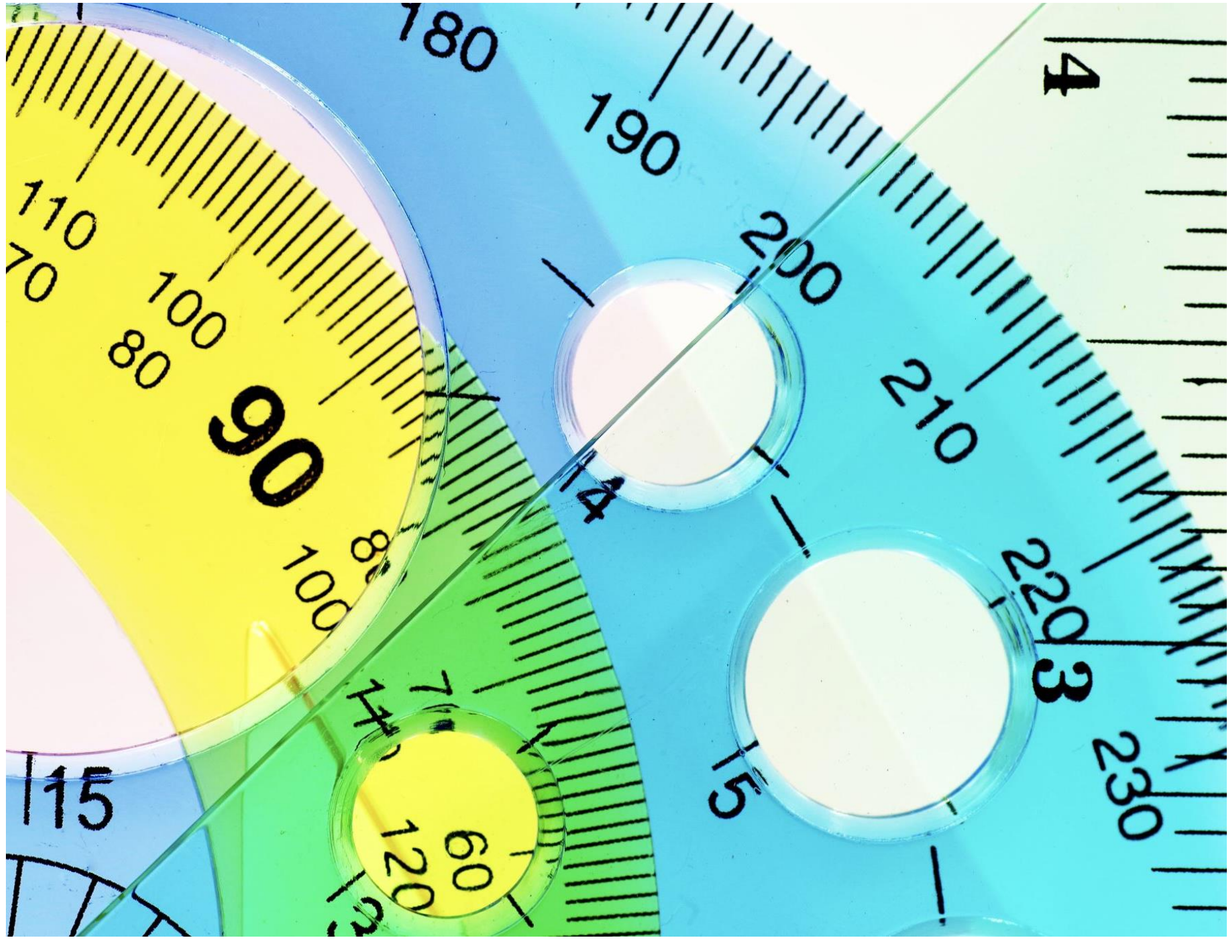
direct measures

- size, effort, schedule, and quality

indirect measures

- measures derived from direct measures, e.g., productivity (amount/unit of time)

Measurement Process



- Formulation. The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
- Collection. The mechanism used to accumulate data required to derive the formulated metrics.
- Analysis. The computation of metrics and the application of mathematical tools.
- Interpretation. The evaluation of metrics results in an effort to gain insight into the quality of the representation.
- Feedback. Recommendations derived from the interpretation of product metrics transmitted to the software team.

Measures

Program Size Measures

- SLOC - source lines of code OR Function Point
- Line of code is typically correlated with effort. Boehm, Walston-Felix, and Halstead all show Effort as a function of lines of code.

Effort Measures

- Most common units of measurement of effort are labour-month, staff week, staff-month, person-year.

Attribute Class	Describes and distinguishes
Type of labour	direct and indirect labor: labor costs that can be charged directly to the project or contract, and those that cannot
hour information	regular or overtime work, and salaried or hourly workers
employment class	regular company employees, whether full-time or part-time, and employees brought in to work on a specific project task, such as consultants and subcontractors
labour class	workers by the types of work they do: managers at various levels, analysts, designers, programmers, documentation specialists, support staff, etc.
Activity	software development activities and maintenance activities
product-level functions	functions of software development, such as design, coding, testing, and documentation; organised by major functional element, by customer release, and by system

Quality Measures

- Two fundamental ideas related to quality are freedom from defect and suitability for use. This suggests quality measures should be counts of defects and problem reports.

Attribute Class	Describes and distinguishes
problem status	points in the problem analysis and correction process: open or closed; recognized, evaluated, or resolved
problem type	software defect or other kind of problem (hardware, operating system, user mistake, operations mistake, new requirement, enhancement); for a software defect, whether it is a defect in requirements, design, code, operational document, test case, etc.
uniqueness	new and unique defect, or a duplicate of another reported defect
criticality	degree of disruption to a user when the problem is encountered
urgency	degree of importance given to the evaluation, resolution, and closure of the problem
finding activity	the activity that uncovered the problem, such as synthesis, inspection, formal review, testing, customer use
finding mode	operational or non-operational environment where defect was found

Performance Measures

- Two common performance measures:
  - response time - how long it takes to accomplish a task
  - throughput - tasks can be completed in a unit of time

Reliability Measures

- Software reliability cannot be measured directly. It is generally computed from other measures of the behavior of the software.
  - Mean Time Between Failures
  - Mean Time to Repair
  - Availability

The Software Quality Metrics Framework

1. Facilitate management control, planning and managerial intervention.
Based on:

- Deviations of actual from planned performance.
- Deviations of actual timetable and budget performance from planned.

2. Identify situations for development or maintenance process improvement (preventive or corrective actions). Based on:

- Accumulation of metrics information regarding the performance of teams, units, etc.

Quality requirements that the software product must meet

Quality factors – Management-oriented attributes of software that contribute to its quality

Quality subfactors – Decompositions of a quality factor to its technical components

Metrics – quantitative measures of the degree to which given attributes (factors) are present

Software Quality Metrics

Desired attributes of Metrics (Ejiogu, 1991)

- Simple and computable
- Empirical and intuitively persuasive
- Consistent and objective
- Consistent in the use of units and dimensions
- Independent of programming language, so directed at models (analysis, design, test, etc.) or structure of program
- Effective mechanism for quality feedback

Subfactors & Direct Metrics

Understandability – The amount of effort required to understand software

Understanding
• Learning time: Time for new user to gain basic understanding of features of the software

Ease of learning – The degree to which user effort required to learn how to use the software is minimized

Ease of learning
• Learning time: Time for new user to learn how to perform basic functions of the software

Operability – The degree to which the effort required to perform an operation is minimized

Operability
• Operation time: Time required for a user to perform operation(s) of the software

Communicativeness – The degree to which software is designed in accordance with the psychological characteristics of users

Communicativeness
• Human factors: Number of negative comments from new users regarding ergonomics, human factors, etc.

Software Quality Metrics

correctness

- defects per KLOC

maintainability

- mean time to change (MTTC) the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users
- spoilage = (cost of change / total cost of system)

integrity

- threat = probability of attack (that causes failure)
- security = probability attack is repelled
- Integrity = Σ [1 - threat * (1 - security)]

Types of Metrics (different classifications)

1

- **Product Metrics**
- **Process Metrics**
- **Project Metrics**

2

Analysis Metrics
- COCOMO
- Function/Feature Points

Design Metrics
- Coupling
- Cohesion
- Structural Complexity
- Data Complexity
- System Complexity

Coding Metrics
- Lines of Code
- McCabe's Cyclomatic Complexity

3

Classification by subjects of measurements
- Quality
- Timetable
- Effectiveness (of error removal and maintenance services)
- Productivity

Types of Metrics – 1st Categorisation

Product Metrics

- focus on the quality of deliverables
- measures of analysis model
- complexity of the design
  - internal algorithmic complexity
  - architectural complexity
  - data flow complexity
- code measures (e.g., Halstead)
- measures of process effectiveness
  - e.g., defect removal efficiency

Process Metrics

- majority focus on quality achieved as a consequence of a repeatable or managed process
- statistical SQA data
  - error categorisation & analysis
- defect removal efficiency
  - propagation from phase to phase
- reuse data

Project Metrics

- Effort/time per SE task
- Errors uncovered per review hour
- Scheduled vs. actual milestone dates
- Changes (number) and their characteristics
- Distribution of effort on SE tasks

Product Metrics

Product Metrics

- Number and type of defects found during requirements, design, code, and test inspections
- Number of pages of documentation delivered
- Number of new source lines of code created
- Number of source lines of code delivered
- Total number or source lines of code delivered
- Average complexity of all modules delivered
- Average size of modules
- Total number of modules
- Total number of bugs found as a result of unit testing
- Total number of bugs found as a result of integration testing
- Total number of bugs found as a result of validation testing
- Productivity, as measured by KLOC per person-hour

The function point method

- The function point approach for software sizing was invented by Allan Albrecht in 1979
- The measure of Albrecht - Function Point Analysis (FPA) - is well known because of its great advantages:
  - Independent of programming language and technology.
  - Comprehensible for client and user.
  - Applicable at early phase of software life cycle.

The function point estimation process:

- **Stage 1**: Compute crude function points **(CFP).**
- **Stage 2**: Compute the relative complexity adjustment factor **(RCAF)** for the project. RCAF varies between 0 and 70.
- **Stage 3**: Compute the number of function points **(FP):**

FP = CFP x (0.65 + 0.01 x RCAF)

Main advantages

- Estimates can be prepared at the pre-project stage.
- Based on requirement specification documents (not specific dependent on development tools or programming languages), the method's reliability is relatively high.

Main disadvantages

- FP results depend on the counting instruction manual.
- Estimates based on detailed requirements specifications, which are not always available.
- The entire process requires an experienced function point team and substantial resources.
- The evaluations required result in subjective results.
- Successful applications are related to data processing. The method cannot yet be universally applied.

Crude function points (CFP) Calculation

The method relates to the following five types of software system components:

- Number of user inputs – distinct input applications, not including inputs for online queries.
- Number of user outputs – distinct output applications such as batch processed reports, lists, customer invoices and error messages (not including online queries).
- Number of user online queries – distinct online applications, where output may be in the form of a printout or screen display.
- Number of logical files – files that deal with a distinct type of data and may be grouped in a database.
- Number of external interfaces – computer–readable output or inputs transmitted through data communication, on CD, diskette, etc.

The function point method applies weight factors to each component according to its complexity.

Software system components	Complexity level									Total CFP
	Simple			average			complex			
	Count	Weight Factor	Points	Count	Weight Factor	Points	Count	Weight Factor	Points	
	A	B	C= AxB	D	E	F= DxE	G	H	I= GxH	
User inputs		3			4			6		
User outputs		4			5			7		
User online queries		3			4			6		
Logical files		7			10			15		
External interfaces		5			7			10		
Total CFP										

Calculating the relative complexity adjustment factor (RCAF)

- The relative complexity adjustment factor (RCAF) summarizes the complexity characteristics of the software system and varies between 0 and 70.
- Assign grades (0 to 5) to the 14 subjects that substantially affect the required development efforts (Extent of distributed processing, performance requirements ...).
- RCAF is the sum of grades regarding the 14 subjects.

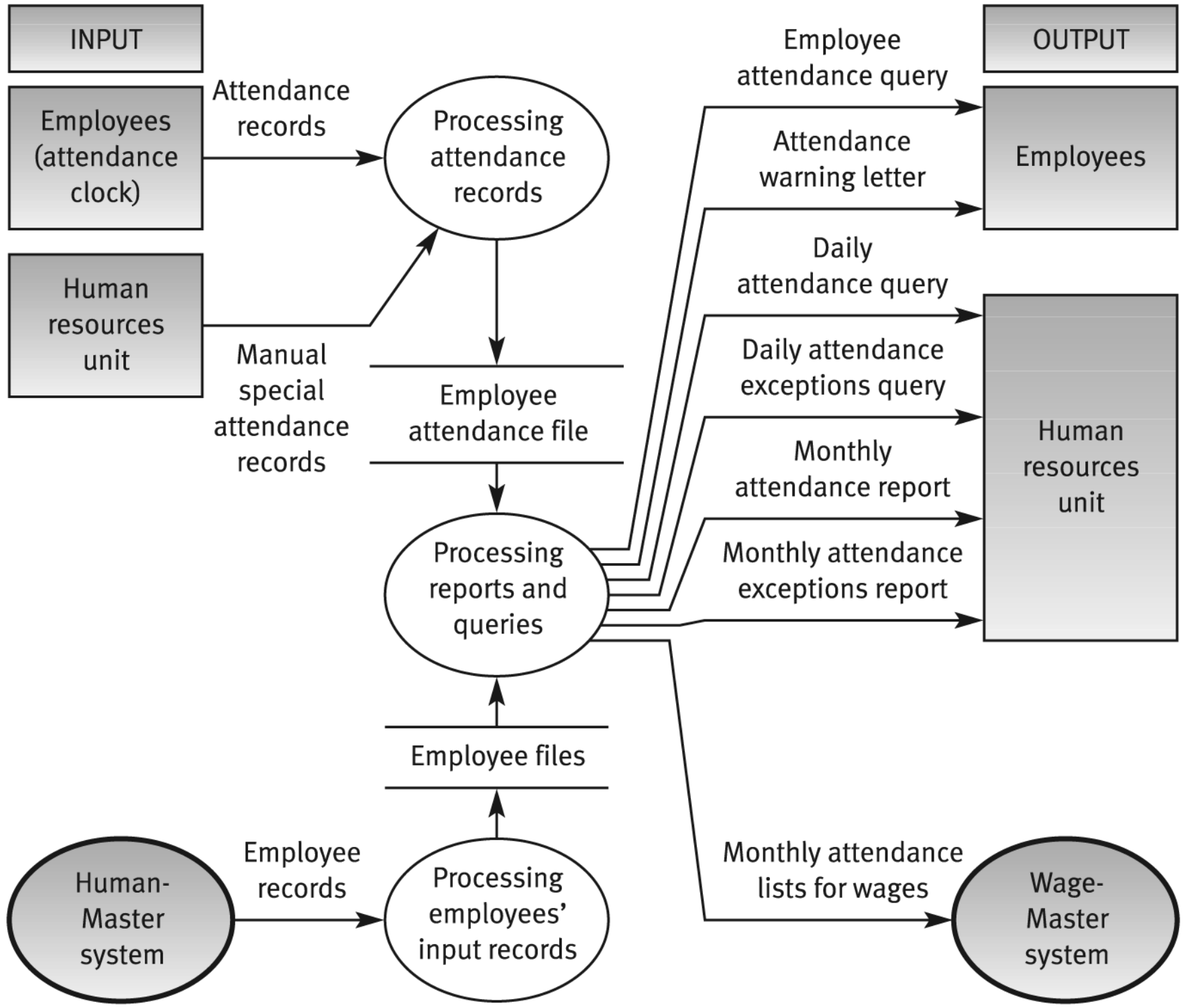
No	Subject	Grade
1	Requirement for reliable backup and recovery	0 1 2 3 4 5
2	Requirement for data communication	0 1 2 3 4 5
3	Extent of distributed processing	0 1 2 3 4 5
4	Performance requirements	0 1 2 3 4 5
5	Expected operational environment	0 1 2 3 4 5
6	Extent of online data entries	0 1 2 3 4 5
7	Extent of multi-screen or multi-operation online data input	0 1 2 3 4 5
8	Extent of online updating of master files	0 1 2 3 4 5
9	Extent of complex inputs, outputs, online queries and files	0 1 2 3 4 5
10	Extent of complex data processing	0 1 2 3 4 5
11	Extent that currently developed code can be designed for reuse	0 1 2 3 4 5
12	Extent of conversion and installation included in the design	0 1 2 3 4 5
13	Extent of multiple installations in an organization and variety of customer organizations	0 1 2 3 4 5
14	Extent of change and focus on ease of use	0 1 2 3 4 5
	Total = RCAF	

Function Point Method

- **An example: The Attend Master**

- *Attend-Master* is a basic employee attendance system that is planned to serve small to medium-sized businesses employing 10–100 employees.
- The system is planned to have interfaces to the company's other software packages: Human-Master, which serves human resources units, and Wage-Master, which serves the wages units.
- Attend-Master is planned to produce several reports and online queries.

The ATTEND MASTER - Data Flow Diagram



Analysis of the software system as presented in the DFD summarises the number of various components:

- Number of user inputs – 2
- Number of user outputs – 3
- Number of user online queries – 3
- Number of logical files – 2
- Number of external interfaces – 2.

The degree of complexity **(simple, average or complex)** was evaluated for each component.

The ATTEND MASTER CFP calculation form

Software system components	Complexity level									Total CFP
	Simple			average			complex			
	Count	Weight Factor	Points	Count	Weight Factor	Points	Count	Weight Factor	Points	
	A	B	C= AxB	D	E	F= DxE	G	H	I= GxH	
User inputs	1	3	3	---	4	---	1	6	6	9
User outputs	---	4	---	2	5	10	1	7	7	17
User online queries	1	3	3	1	4	4	1	6	6	13
Logical files	1	7	7	---	10	---	1	15	15	22
External interfaces	---	5	---	---	7	---	2	10	20	20
Total CFP										81

Calculation of RCAF

Relative Complexity Adjustment Factor

No	Subject	Grade
1	Requirement for reliable backup and recovery	0 1 2 3 4 5
2	Requirement for data communication	0 1 2 3 4 5
3	Extent of distributed processing	0 1 2 3 4 5
4	Performance requirements	0 1 2 3 4 5
5	Expected operational environment	0 1 2 3 4 5
6	Extent of online data entries	0 1 2 3 4 5
7	Extent of multi-screen or multi-operation online data input	0 1 2 3 4 5
8	Extent of online updating of master files	0 1 2 3 4 5
9	Extent of complex inputs, outputs, online queries and files	0 1 2 3 4 5
10	Extent of complex data processing	0 1 2 3 4 5
11	Extent that currently developed code can be designed for reuse	0 1 2 3 4 5
12	Extent of conversion and installation included in the design	0 1 2 3 4 5
13	Extent of multiple installations in an organization and variety of customer organizations	0 1 2 3 4 5
14	Extent of change and focus on ease of use	0 1 2 3 4 5
	Total = RCAF	41

The ATTEND MASTER – function points calculation

FP = CFP x (0.65 + 0.01 x RCAF)

FP = 81 x (0.65 + 0.01 x 41)
= 85.86

Converting NFP to KLOC

- The estimates for the average number of lines of code (LOC) required for programming a function point are the following:

For C++:
KLOC = (85.86 * 64)/1000 = 5.495 KLOC

Programming language/development tool	Average LOC
C	128
C++	64
Visual Basic	32
Power Builder	16
SQL	12

Extended function point metrics Feature Points, UCPs …

- The function point metric was initially designed to be applied to business information systems applications.
- The data dimension was emphasised to the exclusion of the functional and behavioural (control) dimensions.
- The function point measure was inadequate for many engineering and embedded systems

→ **Feature points:** A superset of the function point, designed for applications in which algorithmic complexity is high (real-time, process control, embedded software applications).

→ **UCPs:** Use case points (UCPs) allow the estimation of an application's size and effort from its use cases. UCPs are based on the number of actors, scenarios, and various technical and environmental factors in the use case diagram.

UCPs

- UCPs are based on the number of actors, scenarios, and various technical and environmental factors in the use case diagram.

- The UCP equation is based on four variables:
  - Technical complexity factor (TCF)
  - Environment complexity factor (ECF)
  - Unadjusted use case points (UUCP)
  - Productivity factor (PF)

  which yield the equation:

  UCP = TCP * ECF * UUCP * PF

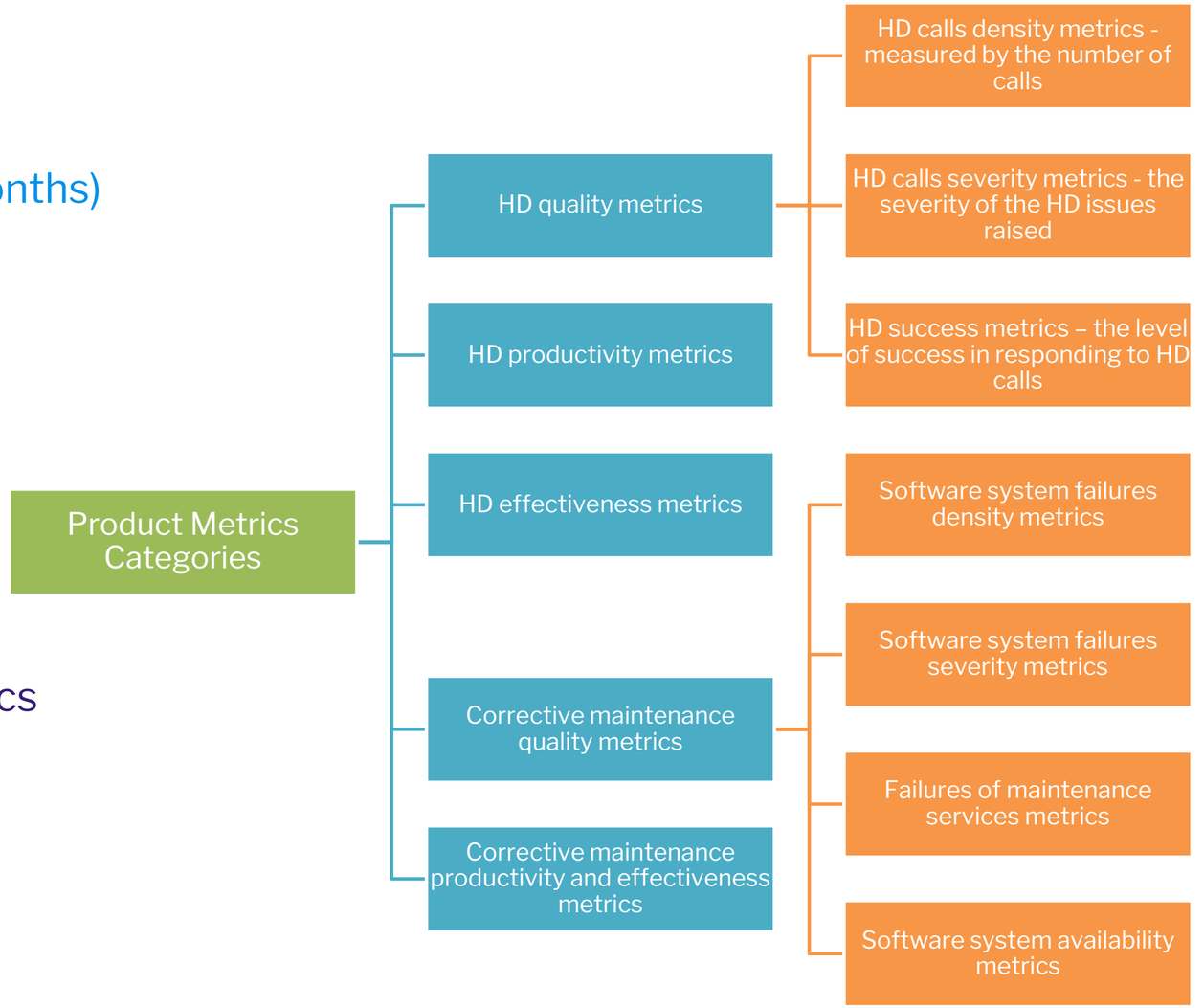
Product Metrics

Refer to Operational Phase
Rely on Performance Reports during a specified period (6-12 months)
Comparison between successive years or different units

- Help Desk Quality metrics
- Help Desk Productivity metrics
- Help Desk Effectiveness metrics
- Corrective maintenance quality metrics
- Corrective maintenance productivity and effectiveness metrics

All customer calls

Failure Reports



Help desk (HD) Quality Metrics

- HD calls density metrics - measured by the number of calls.
- HD calls severity metrics - the severity of the HD issues raised.
- HD success metrics – the level of success in responding to HD calls.

Code	Name	Calculation Formula
HDD	HD calls density	$HDD = \frac{NHYC}{KLMC}$
WHDD	Weighted HD calls density	$WHYC = \frac{WHYC}{KLMC}$
WHDF	Weighted HD calls per function point	$WHDF = \frac{WHYC}{NMFP}$

NHYC = the number of HD calls during a year of service.
KLMC = Thousands of lines of maintained software code.
WHYC = weighted HD calls received during one year of service.
NMFP = number of function points to be maintained.

Code	Name	Calculation Formula
ASHC	Average severity of HD calls	$ASHC = \frac{WHYC}{NHYC}$

NHYC = the number of HD calls during a year of service.
WHYC = weighted HD calls received during one year of service.

Code	Name	Calculation Formula
HDS	HD service success	$HDS = \frac{NHYOT}{NHYC}$

NHYNOT = Number of yearly HD calls completed on time during one year of service.
NHYC = the number of HD calls during a year of service.

Code	Name	Calculation Formula
HDP	HD Productivity	$HDP = \frac{HDYH}{KLNC}$
FHDP	Function Point HD Productivity	$FHDP = \frac{HDYH}{NMFP}$
HDE	HD effectiveness	$HDE = \frac{HDYH}{NHYC}$

HDYH = Total yearly working hours invested in HD servicing of the software system.
KLMC = Thousands of lines of maintained software code.
NMFP = number of function points to be maintained.
NHYC = the number of HD calls during a year of service.

Corrective Maintenance Quality Metrics

- Software system failures density metrics
- Software system failures severity metrics
- Failures of maintenance services metrics
- Software system availability metrics

Code	Name	Calculation Formula
SSFD	Software System Failure Density	$SSFD = \frac{NYF}{KLMC}$
WSSFD	Weighted Software System Failure Density	$WFFFD = \frac{WYF}{KLMC}$
WSSFF	Weighted Software System Failures per Function point	$WSSFF = \frac{WYF}{NMFP}$

NYF = number of software failures detected during a year of maintenance service.
WYF = weighted number of yearly software failures detected during one year of maintenance service.
NMFP = number of function points designated for the maintained software.
KLMC = Thousands of lines of maintained software code.

Code	Name	Calculation Formula
ASSSF	Average Severity of Software System Failures	$ASSSF = \frac{WYF}{NYF}$

NYF = number of software failures detected during a year of maintenance service.
WYF = weighted number of yearly software failures detected during one year.

Code	Name	Calculation Formula
MRepF	Maintenance Repeated repair Failure metric -	$MRepF = \frac{RepYF}{NYF}$

NYF = number of software failures detected during a year of maintenance service.
RepYF = Number of repeated software failure calls (service failures).

Code	Name	Calculation Formula
FA	Full Availability	$FA = \frac{NYSerH - NYFH}{NYSerH}$
VitA	Vital Availability	$VitA = \frac{NYSerH - NYVitFH}{NYSerH}$
TUA	Total Unavailability	$TUA = \frac{NYTFH}{NYSerH}$

NYSerH = Number of hours software system is in service during one year.
NYFH = Number of hours where at least one function is unavailable (failed) during one year, including total failure of the software system.
NYVitFH = Number of hours when at least one vital function is unavailable (failed) during one year, including total failure of the software system.
NYTFH = Number of hours of total failure (all system functions failed) during one year.
$NYFH \geq NYVitFH \geq NYTFH$.
$1 - TUA \geq VitA \geq FA$

Corrective Maintenance Productivity and Effectiveness Metrics

Code	Name	Calculation Formula
CMaiP	Corrective Maintenance Productivity	$CMaiP = \frac{CMaiYH}{KLMC}$
FCMP	Function point Corrective Maintenance Productivity	$FCMP = \frac{CMaiYH}{NMFP}$
CMaiE	Corrective Maintenance Effectiveness	$CMaiE = \frac{CMaiYH}{NYF}$

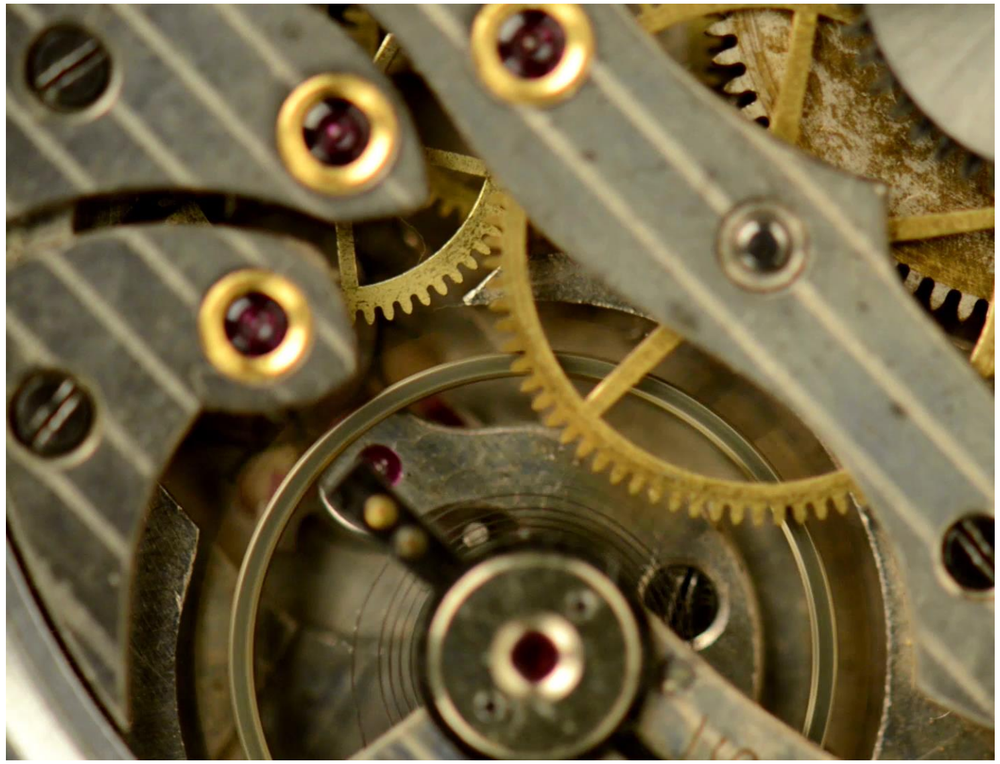
CMaiYH = Total yearly working hours invested in the corrective maintenance of the software system.
NYF = number of software failures detected during a year of maintenance service.
NMFP = number of function points designated for the maintained software.
KLMC = Thousands of lines of maintained software code.

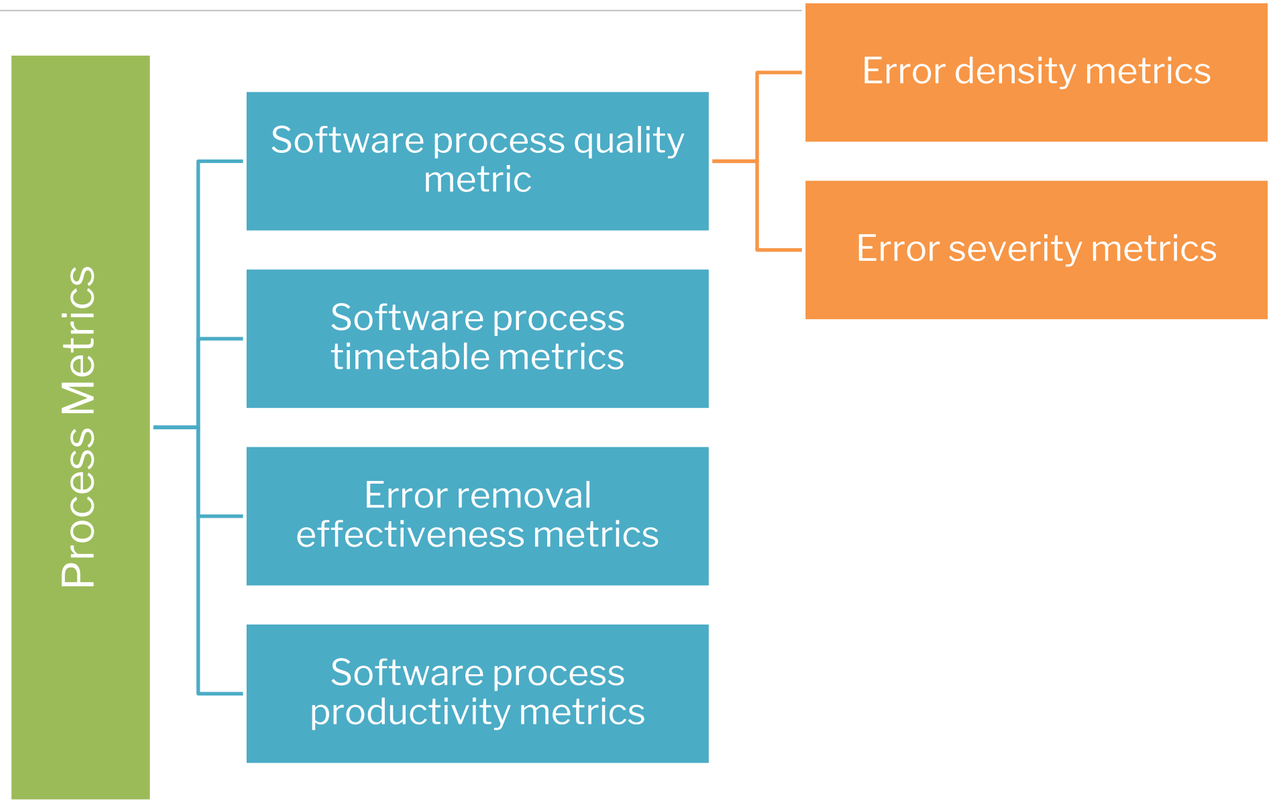
High productivity - less manpower for maintenance

Process Metrics

- Software process quality metrics: error density and severity
- Software process timetable metrics
- Software process error removal effectiveness metrics
- Software process productivity metrics

Process Metrics





- Average find-fix cycle time
- Number of person-hours per inspection
- Number of person-hours per KLOC
- Average number of defects found per inspection
- Number of defects found during inspections in each defect category
- Average amount of rework time
- Percentage of modules that were inspected

Quality Metrics - Error Density Metrics
Measures & Metrics for Error Counting

Number of Code Errors (NCE) vs Weighted Number of Code Errors (WCE)

	Calculation of NCE	Calculation of WCE	
Error severity class	**Number of Errors**	**Relative Weight**	**Weighted Errors**
a	b	c	D = b x c
low severity	42	1	42
medium severity	17	3	51
high severity	11	9	99
Total	**70**	**---**	**192**
NCE	**70**	**---**	**---**
WCE		**---**	**192**

Code	Name	Calculation formula
CED	**Code Error Density**	$CED = \frac{NCE}{KLOC}$
DED	**Development Error Density**	$DED = \frac{NDE}{KLOC}$
WCED	**Weighted Code Error Density**	$WCDE = \frac{WCE}{KLOC}$
WDED	**Weighted Development Error Density**	$WDED = \frac{WDE}{KLOC}$
WCEF	**Weighted Code Errors per Function Point**	$WCEF = \frac{WCE}{NFP}$
WDEF	**Weighted Development Errors per Function Point**	$WDEF = \frac{WDE}{NFP}$

NCE = The number of code errors detected by code inspections and testing.
NDE = total number of development (design and code) errors detected in the development process.
WCE = weighted total code errors detected by code inspections and testing.
WDE = total weighted development (design and code) errors detected in development process.

Error Severity Metrics

Code	Name	Calculation formula
ASCE	Average Severity of Code Errors	$ASCE = \frac{WCE}{NCE}$
ASDE	Average Severity of Development Errors	$ASDE = \frac{WDE}{NDE}$

NCE = The number of code errors detected by code inspections and testing.
NDE = total number of development (design and code) errors detected in the development process.
WCE = weighted total code errors detected by code inspections and testing.
WDE = total weighted development (design and code) errors detected in development process.

When # of errors are generally decreasing, to detect increasing # of severe errors

Timetable Metrics

Code	Name	Calculation formula
TTO	Timetable Observance	$TTO = \frac{MSOT}{MS}$
ADMC	Average Delay of Milestone Completion	$ADMC = \frac{TCDAM}{MS}$

MSOT = Milestones completed on time.
MS = Total number of milestones.
TCDAM = Total Completion Delays (days, weeks, etc.) for all milestones.

to identify

- accounts of success - completion of milestones per schedule
- failure events (non-completion per schedule)
- Average delay in completion per schedule

Error Removal Effectiveness Metrics

Code	Name	Calculation formula
DERE	Development Errors Removal Effectiveness	$DERE = \frac{NDE}{NDE + NYF}$
DWERE	Development Weighted Errors Removal Effectiveness	$DWERE = \frac{WDE}{WDE+WYF}$

NDE = total number of development (design and code) errors) detected in the development process.
WCE = weighted total code errors detected by code inspections and testing.
WDE = total weighted development (design and code) errors detected in development process.
NYF = number software failures detected during a year of maintenance service.
WYF = weighted number of software failures detected during a year of maintenance service.

Can be measured after a period of regular system operation : 6-12 months

Productivity Metrics

Code	Name	Calculation formula
DevP	Development Productivity	$DevP = \frac{DevH}{KLOC}$
FDevP	Function point Development Productivity	$FDevP = \frac{DevH}{NFP}$
CRe	Code Reuse	$Cre = \frac{ReKLOC}{KLOC}$
DocRe	Documentation Reuse	$DocRe = \frac{ReDoc}{NDoc}$

DevH = Total working hours invested in the development of the software system.
ReKLOC = Number of thousands of reused lines of code.
ReDoc = Number of reused pages of documentation.
NDoc = Number of pages of documentation.

Deal with human resource productivity & indirectly extent of software reuse

Project Metrics

Project Metrics - Monitoring & Control

Successful monitoring and control depends on accurate and current project work performance information

Daily raw numbers – time expended per task, cost information, milestones met

Frequency numbers – bugs per, user issues reported per

Qualitative assessments – user reported likes/dislikes with product, team member reported task percent complete estimates

Metrics

Estimated time to completion

Budget at completion

Impact on customers

...

Project Metrics Schedule Control

is concerned with:

- determining the current status of all items currently being worked on,
- influencing factors that create schedule changes,
- determining that the schedule has or has not changed, and
- managing changes to the schedule using a formal Integrated Change Control process

Potential information to collect:

- milestones achieved on time and on budget,
- hours worked on each task,
- hours remaining to complete each active task,
- resource availability issues such as turnover or health issues,
- cost information for resources and other budget items,
- risk information,
- quality information,
- scope changes, and
- vendor issues

Project Metrics Milestone Analysis

Milestones are *events or stages* of the project that represent a *significant accomplishment.*

Milestones

- … **show completion** of important steps
- … **signal** the team and suppliers
- … can **motivate** the team
- … offer **reevaluation** points
- … help **coordinate** schedules
- … **identify** key review gates
- … **delineate** work packages

Project Metrics
Cost Control

- is concerned with:
  - influencing the factors that create cost variances on the project and
  - controlling changes to the project's budget
- Like the other monitoring and control processes, cost control is a continual process of comparing the current actual project expenditures to the defined budget and determining when issues have arisen that need to be dealt with
- *Almost every change made on an IT project will affect Cost in some manner*

Project Metrics
Earned Value Management (EVM) or Earned Value Analysis (EVA)

- Earned value
  - is a measure of progress
  - enables us to assess the "percent of completeness" of a project using quantitative analysis rather than rely on a gut feeling
  - "provides accurate and reliable readings of performance from as early as 15 percent into the project." [FLE98]
  - A technique used to help determine and manage project progress and the magnitude of any variations from the planned values concerning cost, schedule, and performance
- The technique was created to help the project team and stakeholders gain a better understanding of just how the project is performing
- Many project managers fail to evaluate performance properly
  - How much work has actually been completed and how much work actually remains
  - Not necessarily how many hours have been worked

Project Metrics – EVA/EVM

- **Percentage of Completion (PoC)=** Rate of performance
  - Often IT projects can be difficult to estimate progress
  - 0-100 percent rule
  - 50-50 percent rule
  - Interval percent rule (0, 25, 50, 75, 100)
- **Planned Value (PV)** – is the budgeted cost for the work scheduled to be completed on a task, work package, or activity up to a given point in time (BCWS)
- **Actual Cost (AC)** – is the total cost incurred in accomplishing work on the task during a given time period (ACWP)
- **Earned Value (EV)** – is the budgeted amount for the work actually completed on the task during a given time period (BCWP) or

  EV = (PV)*(percent complete)
- **Cost Variance (CV)** – equals earned value (EV) minus actual cost (AC) or

  CV = EV – AC
- **Schedule Variance (SV)** - equals earned value (EV) minus planned value (PV) or

  SV = EV – PV
- **Cost Performance Index (CPI)** - equals the ratio of EV to the AC, or

  CPI = EV/AC
  - Equal to 100% then Actual = Planned
  - Less than 100% then project is over budget
- **Schedule Performance Index (SPI)** – equals the ratio of EV to the PV, or

  SPI = EV/PV
  - Equal to 100% then Actual = Planned
  - Less than 100% project is behind schedule
- **Budget at Completion (BAC)**
  - *How much did you BUDGET for the Total Job?*
  - BAC = $\Sigma (PV_k)$ for all tasks *k*
- **Estimate at Completion (EAC)**
  - *What do we currently expect the TOTAL project to cost?*

  EAC = BAC/CPI
- **Estimate to Complete (ETC)**
  - *From this point on, how much MORE do we expect it to cost to finish the job?*

  ETC = EAC - AC

Analysis, Design & Coding Metrics

Analysis Metrics

- **Function-based metrics:** use the function point as a normalizing factor or as a measure of the “size” of the specification
- **COCOMO:** COnstructive COst MOdel (COCOMO) is an algorithmic Software Cost Estimation Model developed by Barry Boehm
- Bang metric: used to develop an indication of software “size” by measuring characteristics of the data, functional and behavioral models
- Specification metrics: used as an indication of quality by measuring number of requirements by type

Normalised data are used to evaluate the process and the product (but never individual people)

size-oriented normalisation — the line of code approach

function-oriented normalisation — the function point approach

Analysis Metrics - ...

Typical Function-Oriented Metrics

- $ per FP
- pages of documentation per FP
- FP per person-month
- :
- errors per FP
- defects per FP

Analysis

Typical Size-Oriented Metrics

- page of documentation per KLOC
- LOC per person-month
- $ / page of documentation
- $ per LOC
- :
- errors per KLOC (thousand lines of code)
- defects per KLOC
- errors / person-month

Analysis

Design Metrics

Architectural Design Metrics

- **Structural complexity** = g(fan-out)
- **Data complexity** = f(input & output variables, fan-out)
- **System complexity** = h(structural & data complexity)
- **HK metric**: architectural complexity as a function of fan-in and fan-out
- **Morphology metrics**: a function of the number of modules and the number of interfaces between modules

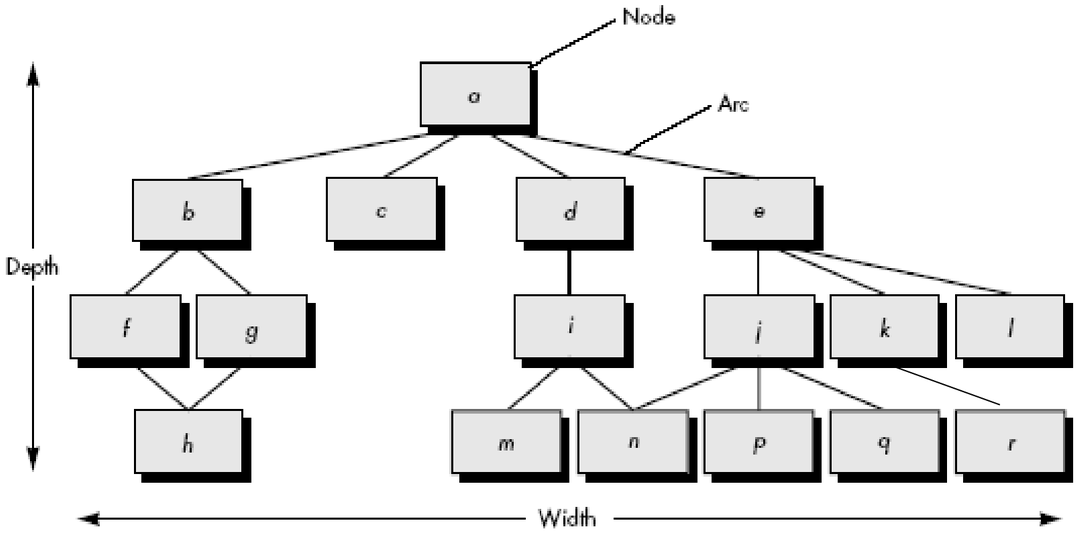
Component-Level Design Metrics

- **Cohesion metrics**: a function of data objects and the locus of their definition
- **Coupling metrics**: a function of input and output parameters, global variables, and modules called
- **Complexity metrics**: hundreds have been proposed (e.g., cyclomatic complexity)

Interface Design Metrics

- **Layout appropriateness**: a function of layout entities, the geographic position and the "cost" of making transitions among entities

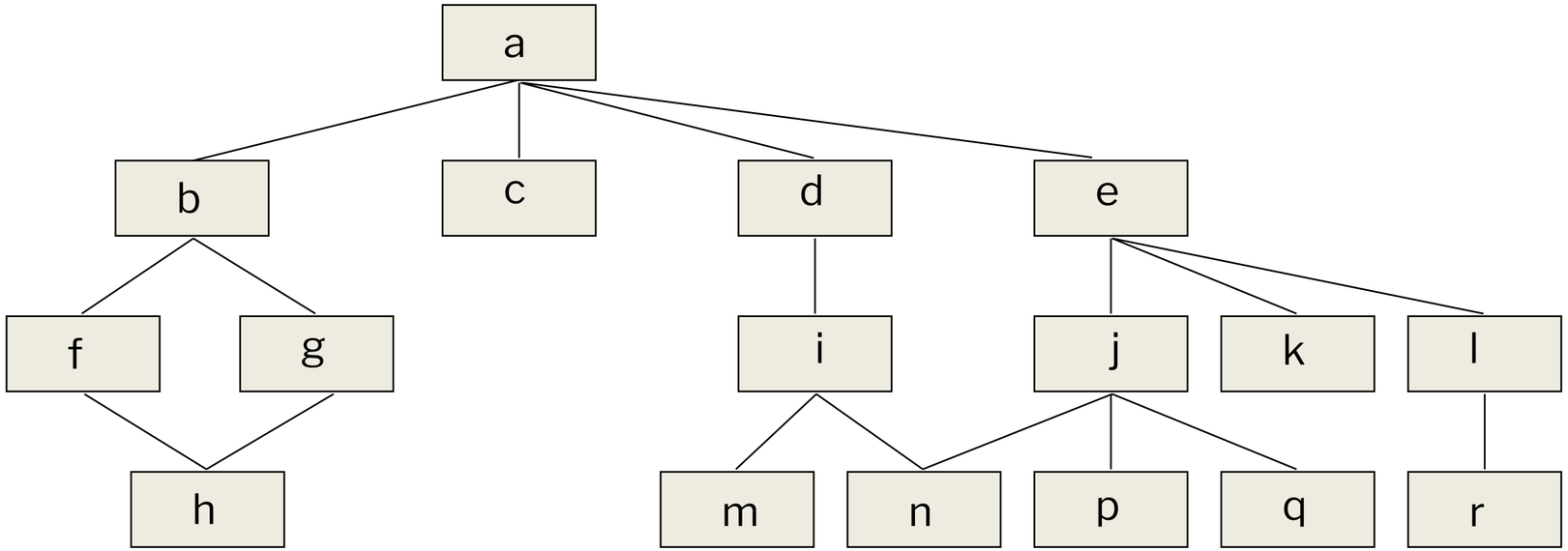
Architectural Design Metrics



- **Structural complexity = g(fan-out)**
  → $S(i) = f^2_{out}(i)$
  where $f_{out}(i)$ is the fan-out of module *i*
- **Data complexity = f(input & output variables, fan-out)**
  → $D(i) = v(i)/[f_{out}(i) + 1]$
  where *v(i)* is the number of input and output variables that are passed to and from module *i.*
- **System complexity = h(structural & data complexity)**
  → $C(i) = S(i) + D(i)$
- **HK metric: architectural complexity as a function of fan-in and fan-out**
  → $HKM = length(i)\ [f_{in}(i) + f_{out}(i)]^2$
  where length*(i)* is the number of programming language statements in a module *I* and $f_{in}(i)$ is the fan-in of a module *i.*
- **Morphology metrics: a function of the number of modules and the number of interfaces between modules**
  → $size = n + a$
  where *n* is the number of nodes and *a* is the number of arcs.

Morphology Metrics

- size = n + a
- n = number of modules
- a = number of arcs (lines of control)
- arc-to-node ratio, r = a/n
- depth = longest path from the root to a leaf
- width = maximum number of nodes at any level



size: **17 + 18** depth:**4** width: **6** arc-to node ratio: **~1**

Component-Level Design Metrics

Metric Type	Measures	Based On	Goal
Cohesion Metrics	Internal unity of a module	Shared data & responsibilities	High cohesion
Coupling Metrics	Dependencies between modules	Parameters, globals, calls	Low coupling
Complexity Metrics	Difficulty of understanding/testing	Control flow, operators, structure	Low complexity

- Cohesion Metrics
- Coupling Metrics
  - data and control flow coupling
  - global coupling
  - environmental coupling
- Complexity Metrics
  - Cyclomatic complexity
  - Experience shows that if this > 10, it is very difficult to test

Coupling Metrics

Data and control flow coupling

- d_i = number of input data parameters
- c_i = number of input control parameters
- d_0 = number of output data parameters
- c_0 = number of output control parameters

Global coupling

- g_d = number of global variables used as data
- g_c = number of global variables used as control

Environmental coupling

- w = number of modules called (fan-out)
- r = number of modules calling the module under consideration (fan-in)
- Module Coupling: $m_c = 1/ (d_i + 2*c_i + d_0 + 2*c_0 + g_d + 2*g_c + w + r)$
- $m_c = 1/(1 + 0 + 1 + 0 + 0 + 0 + 1 + 0) = .33$ (Low Coupling)
- $m_c = 1/(5 + 2*5 + 5 + 2*5 + 10 + 0 + 3 + 4) = .02$ (High Coupling)

Coupling metrics measure **how dependent one module is on other modules**. It quantifies the degree of interaction between modules. Coupling is determined by assessing how many *external connections* a module has, incuding:

1. **Input parameters** it receives
2. **Output parameters** it returns
3. **Global variables** it reads or modifies
4. **Other modules it calls**
5. **How many modules call it**

A module has **low coupling** when it is self-contained and interacts with few external modules.
A module has **high coupling** when it depends heavily on:
- large parameter lists
- shared global variables
- many other modules

Types of Coupling (from tightest to loosest):
- Content coupling (worst)
- Common coupling
- External coupling
- Control coupling
- Stamp coupling
- **Data coupling (best)**
- No coupling (ideal)

Examples of Coupling Metrics:
- **Fan-in / Fan-out** (number of modules calling/called by a module)
- **Coupling Between Objects (CBO)**
- **Message Passing Coupling (MPC)**

Quality Meaning: Low coupling → better maintainability, independence, reusability, and testability

Cohesion Metrics

Cohesion metrics measure **how strongly related and focused the responsibilities of a single module (e.g., a class, function, or component) are.** In software quality, cohesion reflects a module's internal strength.

Cohesion is evaluated by analysing:
- **Data objects** used in the module (variables, data structures).
- **Where these data objects are defined** (their *locus of definition*).

A module has **high cohesion** when:
- Its responsibilities focus on a single task.
- All functions within the module operate on the same set of internal data.

A module has **low cohesion** when:
- Its functions operate on unrelated data objects.
- Responsibilities are mixed (e.g., doing calculations, printing, saving to file).

Types of Cohesion (from weakest to strongest):
- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- **Functional (best)**

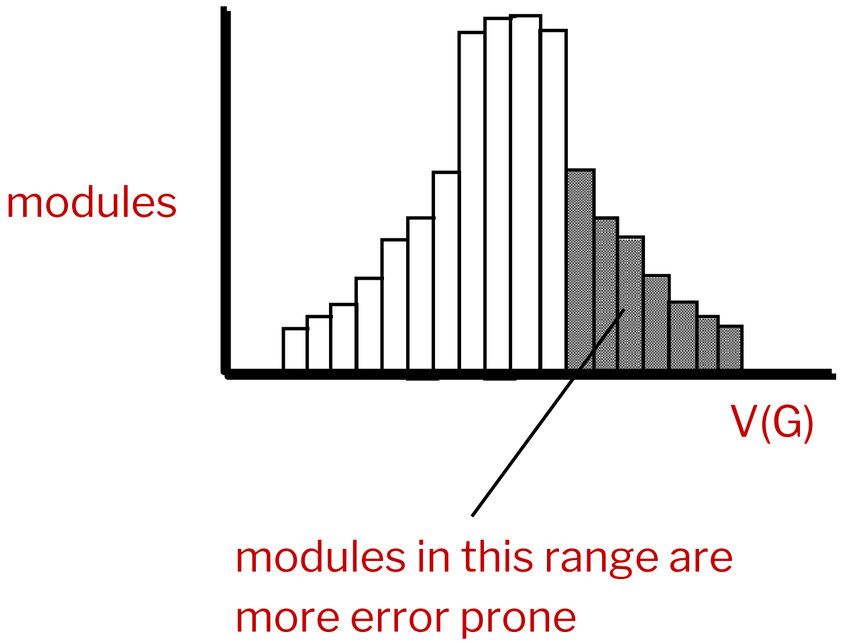
Examples of Cohesion Metrics:
- **LCOM (Lack of Cohesion in Methods):** Counts pairs of methods that do *not* share common data attributes.
- **Tight Class Cohesion (TCC)**
- **Loose Class Cohesion (LCC)**

Quality Meaning: Higher cohesion → better modularity, easier maintenance, fewer side effects, clearer responsibility.

Metrics for Source Code

- Cyclomatic Complexity (McCabe)

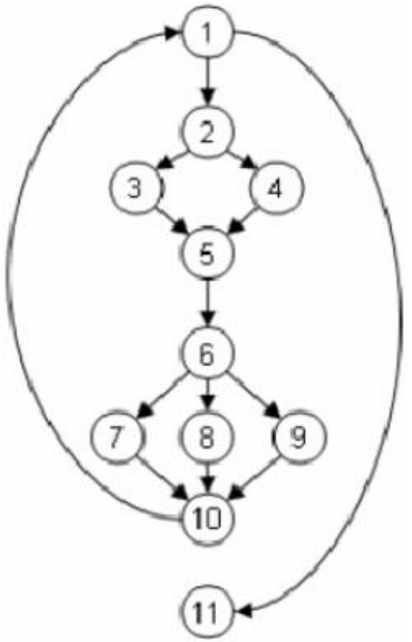
Industry studies have indicated that the higher V(G), the higher the probability or errors.



Measures the number of **independent execution paths** through a program.
Formula:
V(G) = E – N + 2
where:
- **E** = number of edges
- **N** = number of nodes
- **2** = number of connected components

High V(G) → more branches → more test cases required.

Node	Statement
(1)	`while(x<100){`
(2)	`if (a[x] % 2 == 0) {`
(3)	`parity = 0;`
	`}`
	`else {`
(4)	`parity = 1;`
(5)	`}`
(6)	`switch(parity){`
	`case 0:`
(7)	`println( "a[" + i + "] is even");`
	`case 1:`
(8)	`println( "a[" + i + "] is odd");`
	`default:`
(9)	`println( "Unexpected error");`
	`}`
(10)	`x++;`
	`}`
(11)	`p = true;`



Metrics for Source Code

- Maurice HALSTEAD's Software Science
  - n_1 = the number of distinct operators
  - n_2 = the number of distinct operands
  - N_1 = the total number of operator occurrences
  - N_2 = the total number of operand occurrences

Length: $N = N_1 + N_2$

Volume: $V = N\log_2(n_1 + n_2)$

```
Z = 20;
Y = -2;
X = 5;

While  X>0

   Z = Z + Y;

  if  Z > 0 then
      X = X - 1;
   end-if;

End-while;

Print(Z);
```

OPERATOR	COUNT	OPERAND	COUNT
IF-Then- end if	1	Z	5
While End-While	1	Y	2
=	5	X	4
;	8	20	1
>	2	-2	1
+	1	5	1
-	1	0	2
print	1	1	1
( )	1		

$n_1 = 9$ $N_1 = 21$ Length: $N = 21+17 = 38$

$n_2 = 8$ $N_2 = 17$ Volume: $V = 38 \log_2(17)=155$

Metrics for OO Design

Whitmire [WHI97] describes nine distinct and measurable characteristics of an OO design:

- Size: Size is defined in terms of four views: population, volume, length, and functionality
- Complexity: How classes of an OO design are interrelated to one another
- Coupling: The physical connections between elements of the OO design
- 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.”
- Completeness: An indirect implication about the degree to which the abstraction or design component can be reused.
- Cohesion: The degree to which all operations working together to achieve a single, well-defined purpose
- Primitiveness: Applied to both operations and classes, the degree to which an operation is atomic
- Similarity: The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose
- Volatility: Measures the likelihood that a change will occur

Distinguishing Characteristics

- Localisation—the way in which information is concentrated in a program
- Encapsulation—the packaging of data and processing
- Information hiding—the way in which a secure interface hides information about operational details
- Inheritance—the manner in which the responsibilities of one class are propagated to another
- Abstraction—the mechanism that allows a design to focus on essential details

Berard [BER95] argues that the following characteristics require that special OO metrics be developed

Class-Oriented Metrics

Proposed by Chidamber and Kemerer

- weighted methods per class
- depth of the inheritance tree
- number of children
- coupling between object classes
- response for a class
- lack of cohesion in methods

Proposed by Lorenz and Kidd [LOR94]:

- class size
- number of operations overridden by a subclass
- number of operations added by a subclass
- specialisation index

The MOOD Metrics Suite

- Method inheritance factor
- Coupling factor
- Polymorphism factor

Metrics for Testing

- Analysis, design, and code metrics guide the design and execution of test cases.

- Metrics for Testing Completeness
  - Breadth of Testing - total number of requirements that have been tested
  - Depth of Testing - percentage of independent basis paths covered by testing versus total number of basis paths in the program.

Metrics for Maintenance

- Software Maturity Index (SMI)
  - M_T = number of modules in the current release
  - F_c = number of modules in the current release that have been changed
  - F_a = number of modules in the current release that have been added
  - F_d = number of modules from the preceding release that were deleted in the current release

$$SMI = [M_T - (F_c + F_a + F_d)] / M_T$$

- A simple measure of reliability is *mean-time-between-failure* (MTBF), where

  MTBF = MTTF + MTTR

- The acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to-repair*, respectively.
- *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as

  Availability = [MTTF/(MTTF + MTTR)] x 100%

Metrics Derived from Reviews

- inspection time per page of documentation
- inspection time per KLOC or FP
- inspection effort per KLOC or FP
- errors uncovered per reviewer hour
- errors uncovered per preparation hour
- errors uncovered per SE task (e.g., design)
- number of minor errors (e.g., typos)
- number of major errors
  (e.g., nonconformance to req.)
- number of errors found during preparation

General limitations of quality metrics

* Budget constraints in allocating the necessary resources.

* Human factors, especially opposition of employees to evaluation of their activities.

* Validity Uncertainty regarding the data's, partial and biased reporting.

Examples of software metrics that exhibit severe weaknesses

- Parameters used in development process metrics:
  KLOC, NDE, NCE.
- Parameters used in product (maintenance) metrics:
  KLMC, NHYC, NYF.

Factors affecting parameters used for development process metrics

a. Programming style **(KLOC).**
b. Volume of documentation comments **(KLOC).**
c. Software complexity **(KLOC, NCE).**
d. Percentage of reused code **(NDE, NCE).**
e. Professionalism and thoroughness of design review and software testing teams: affects the number of defects detected **(NCE).**
f. Reporting style of the review and testing results: concise reports vs. comprehensive reports **(NDE, NCE).**

Factors affecting parameters used for product (maintenance) metrics

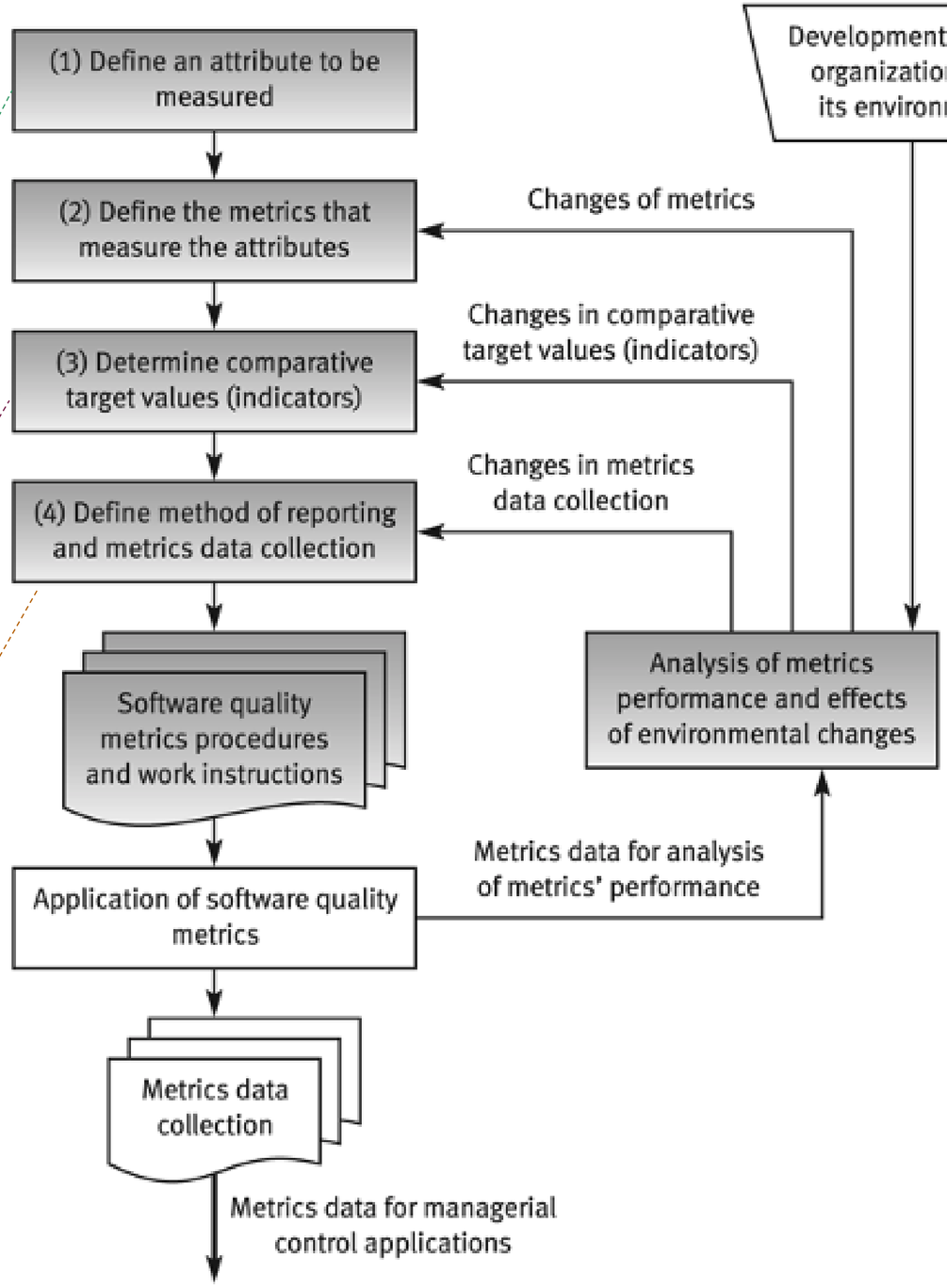
a. Quality of installed software and its documentation **(NYF, NHYC).**
b. Programming style and volume of documentation comments included in the code be maintained **(KLMC).**
c. Software complexity **(NYF).**
d. Percentage of reused code **(NYF).**
e. Number of installations, size of the user population and level of applications in use: **(NHYC, NYF).**

Process of defining software quality metrics

Software quality, development team productivity, etc

Target values: standards, previous year's performance, etc.

Reporting process, frequency of reporting, method(s) of metrics data collection



Discussion Question - 1

You want to track the progress of your team and identify potential risks in meeting deadlines.

- ➔ Number of Requirements Implemented: Tracks how many user stories or features have been coded.
- ➔ Effort Spent per Requirement: Tracks the time developers spend on individual requirements.

You find that 80 out of 100 requirements have been implemented so far, but the time spent on each is increasing compared to earlier phases. This indicates …?

Discussion Question - 2

You want to evaluate how effectively the team is converting resources into deliverables while ensuring quality.

- ➔ Defect Removal Efficiency (DRE): DRE=(Defects Found and Fixed During Development/ Total Defects)×100 This metric shows how effective your testing process is.
- ➔ Code Churn Rate: Measures how much code is added, modified, or deleted over time. %10-20 may be acceptable. High churn rates may indicate unstable requirements or poor initial design.

Your processing metrics reveal:

- A DRE of 70%, indicates ….?
- %35 code churn rate, signals …?

Your processing metrics reveal:

- A DRE of 70%, indicating 30% of defects are slipping past your testing phase.
- A high code churn rate, signaling significant rework due to changing requirements.

If metrics highlight low DRE (Defect Removal Efficiency), you might decide to improve your testing process or allocate more resources to testing.

If metrics show delays in requirement implementation, you could address inefficiencies or scope changes causing the delay.

Backup Slides

Software product metrics

Software metric	Description
Fan in/Fan-out	Fan-in is a measure of the number of functions or methods that call some other function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

Object-oriented metrics

Object-oriented metric	Description
Depth of inheritance tree	This represents the number of discrete levels in the inheritance tree where sub-classes inherit attributes and operations (methods) from super-classes. The deeper the inheritance tree, the more complex the design. Many different object classes may have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described above and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external methods.
Weighted methods per class	This is the number of methods that are included in a class weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as super-classes in an inheritance tree.
Number of overriding operations	This is the number of operations in a super-class that are over-ridden in a sub-class. A high value for this metric indicates that the super-class used may not be an appropriate parent for the sub-class.

Coupling and Cohesion

Goal: Reduction of *complexity while change occurs*

Cohesion measures the dependence among classes

High cohesion: The classes in the subsystem perform similar tasks and are related to each other (via associations) GOOD!

Low cohesion: Lots of miscellaneous and auxiliary classes, no associations BAD!!

Coupling measures dependencies between subsystems

High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.) BAD!!

Low coupling: A change in one subsystem does not affect any other subsystem GOOD!!

Subsystems should have as maximum cohesion and minimum coupling as possible:

How can we achieve high cohesion?

How can we achieve loose coupling?

Coupling

Indicates the interdependence or interrelationships of the modules

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite, (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

The Law of Demeter

- An object should only send messages to one of the following:
  - Itself
  - An object that is contained in an attribute of the object or its superclass
  - An object that is passed as a parameter to the method
  - An object that is created by the method
  - An object that is stored in a global variable

Cohesion

the degree to which a module performs one and only one function.

Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., Calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is send by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

Ideal Class Cohesion

1

Contain multiple methods that are visible outside the class

2

Have methods that refer to attributes or other methods defined with the class or its superclass

3

Not have any control-flow coupling between its methods

Types of Class Cohesion

Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class.
	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

Other Code Metrics

- **Halstead's Software Science:** a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program
- **Lines of Code**
- **McCabe's Cyclomatic Complexity**

Operation-Oriented
Proposed by Lorenz and Kidd [LOR94]:

- average operation size
  operation complexity
- average number of parameters per operation

Project Metrics
Proposed by Lorenz and Kidd [LOR94]:

- number of scenario scripts
- number of key classes
- number of subsystems

Testability Metrics
Proposed by Binder [BIN94]:

- encapsulation related
  - lack of cohesion in methods
  - percent public and protected
  - public access to data members
- inheritance related
  - number of root classes
  - fan in
  - number of children and depth of inheritance tree

Halstead's Software Science

1. Program Vocabulary (n)	The total number of distinct operators and operands: $n = n_1 + n_2$
2. Program Length (N)	Total occurrences of operators and operands: $N = N_1 + N_2$
3. Program Volume (V)	Represents the *size* of the implementation in terms of information content: $V = N \cdot \log_2(n)$
4. Program Difficulty (D)	Indicates how hard the program is to write or understand based on its operators and operands: $D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}$
5. Effort(E)	The mental effort required to implement or understand the program: $E = V \cdot D$
6. Time to Program (T)	Estimated time (in seconds) to write the program: $T = \frac{E}{18}$ (Assuming 18 mental operations per second)
7. Estimated Number of Bugs (B)	Halstead's prediction of delivered bugs: $B = \frac{E^{2/3}}{3000}$