UNIT III:

Technical Metrics For Software-Software Process and Project Metrics- Size Oriented Metrics- Function-Oriented Metrics- Extended Function Point Metrics- A Framework for Technical Software Metrics- Metrics for Requirement Specification Quality- Metrics for Analysis- Metrics for Design- Metrics for Source Code- Metrics for Testing- Metrics for Maintenance. Technical Metrics For Object-Oriented Systems-Intent of Object-Oriented Metrics- Characteristics of Object-Oriented Metrics - Metrics for OO Design Model- Class-Oriented Metrics- Operation-Oriented Metrics- Metrics for Object-Oriented Testing- Metrics for Object-Oriented Projects.

3.1.Technical Metrics For Software:

Metrics is a quantitative measure of the degree to which a system, component, or **process** possesses a given attribute. Measures, **Metrics**, and Indicators. An indicator is a **metric** or combination of **metrics** that provide insight into the **software process**, a **software** project, or the **product** itself.

Software Metrics.

Following are the needs for the software Metrics-

To **characterize** in order to

Gain an understanding of processes, products, resources, and environments.

Establish baselines for comparisons with future assessments

To **evaluate** in order to

Determine status with respect to

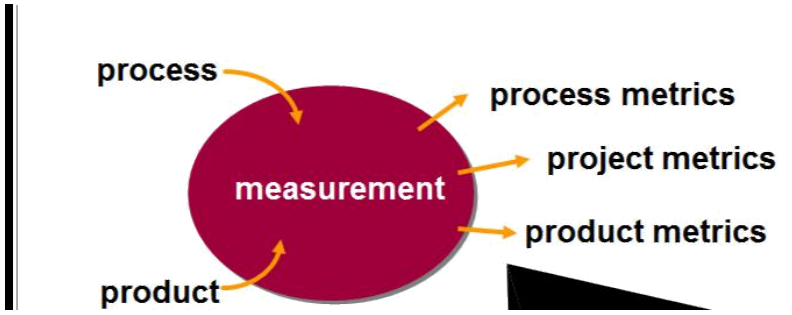
plans To **predict** in order to

Gain understanding of relationships among processes and products.

Build models of these relationships

To **improve** in order to

Identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance.



3.2.Software Process and Project Metrics

Software metrics is a standard of measure that contains many activities which involve some degree of measurement. It can be classified into three categories: product metrics, process metrics, and project metrics.

Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level.

Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process.

Project metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics.

Process Metrics

Process metrics are collected across all projects and over long periods of time.

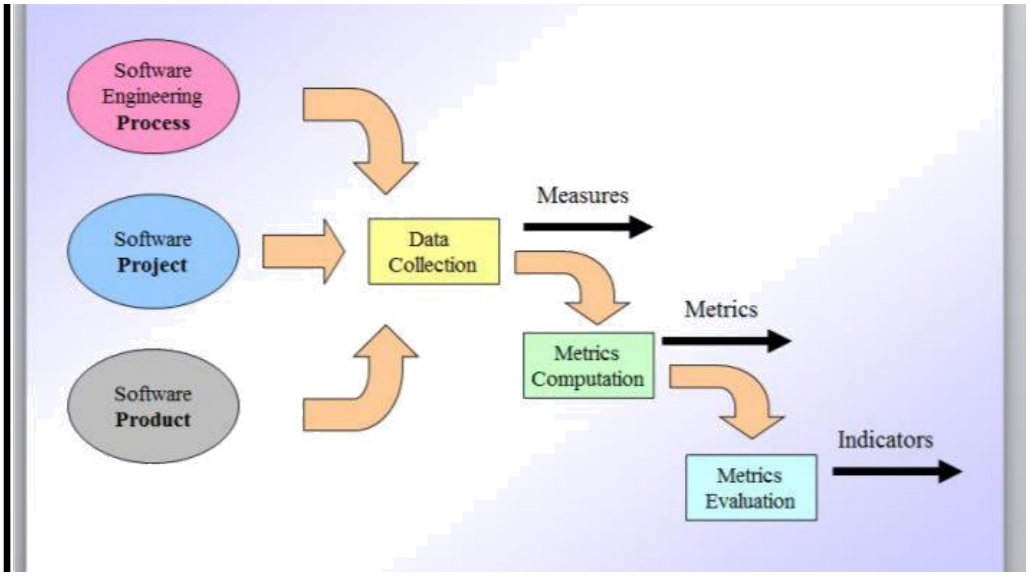
They are used for making strategic decisions.

The intent is to provide a set of process indicators that lead to long-term software process improvement.

The only way to know how/where to improve any process is to

1. Measure specific attributes of the process.
2. Develop a set of meaningful metrics based on these attributes.

3. Use the metrics to provide indicators that will lead to a strategy for improvement.



Product Metrics

They focus on the quality of deliverables. Product metrics are combined across several projects to produce process metrics.

Metrics for the product:

Measures of the Analysis Model.

Complexity of the Design Model

Code metrics.

Furthermore, Complexity of the Design Model is classified as-

1. Internal algorithmic complexity.
2. Architectural complexity.
3. Data flow complexity.

3.3.Size-oriented Metrics

Size-oriented metrics are derived by normalizing the productivity measures considering the size of software that has been produced.

Productivity = KLOC / person-month, where: KLOC- thousand lines of code

Quality = defects / KLOC

Cost = $ / KLOC

Documentation = pages of documentation / KLOC

Attempt to quantify software projects by using the size of the project to normalize other quality measures

Possible data to collect:

number of lines of code

number of person-months to complete

cost of the project

number of pages of documentation

number of errors corrected before release

number of bugs found post release

3.4.Function-Oriented Metrics

Function-oriented metrics use as a normalization value the functionality delivered by the application. Since the functionality can not be measured directly, estimates called function points are derived based on countable information in the software domain and assessments of the software complexity.

Five information domain characteristics are determined as follows:

- number of user inputs: each user input provides distinct application data;

- number of user outputs: each output provides different information to

the user, like reports, screens and messages;

- number of user inquiries: these are on-line inputs that require immediate

response from the software program;

- number of files: each file is counted;

- number of external interfaces: all machine readable interfaces are counted.

Attempt to measure the functionality of a software system

Use a unit of measure called function point

Some possible function points:

Internal data structures

External data structures

User inputs

User outputs

Transformations

Transitions

Issues with Using Function-Oriented Metrics

Requires that analysis and design of a project are completed before workload estimation can occur

Validity of the workload estimation is limited to the accuracy of the analysis and design

Complexity determination of function points is subjective

3.5.Extended Function Point Metrics:

A number of extensions to the basic function-point measure have been proposed to make it more adequate for systems engineering applications.

The feature-point measure is a superset of the basic function-point measure especially for application requiring complex software.

The feature-point measure count another software characteristic: algorithms in addition to the basic information domain characteristics.

3.6.A Framework for Technical Software Metrics :

- As we noted in the introduction to this chapter, measurement assigns numbers orsymbols to attributes of entities in the real word. To accomplish this, a measurementmodel encompassing a consistent set of rules is required.
- The fundamental framework and a set of basic principles for the measurementof technical metrics for softwarewere established.

The Challenge of Technical Metrics

- Over the past three decades, many researchers have attempted to develop a singlemetric that provides a comprehensive measure of software complexity.
- Fenton characterizes this research as a search for "the impossible holy grail." Although dozensof complexity measures have been proposed, each takes a somewhat differentview of what complexity is and what attributes of a system lead to complexity.
- By analogy, consider a metric for evaluating an attractive car. Some observersmight emphasize body design, others might consider mechanical characteristics, stillothers might tout cost, or performance, or fuel economy, or the ability to recycle whenthe car is junked. 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 occurswith computer software.
- Yet there is a need to measure and control software complexity. And if a singlevalue of this quality metric is difficult to derive, it should be possible to develop measuresof different internal program attributes (e.g., effective modularity, functionalindependence, and other attributes.
- These measuresand the metrics derived from them can be used as independent indicators ofthe quality of analysis and design models. But here again, problems arise. Fentonnotes this when he states:The danger of attempting to find measures which characterize so many different attributesis that inevitably the measures have to satisfy conflicting aims. This is counter to the representationaltheory of measurement.
- Although Fenton's statement is correct, many people argue that technical measurementconducted during the early stages of the software process provides softwareengineers with a consistent and objective mechanism for assessing quality.
- It is fair to ask, however, just how valid technical metrics are. That is, how closelyaligned are technical metrics to the long-term reliability and quality of a computerbased system? Fenton addresses this question in the following way:In spite of the intuitive connections between the internal structure of software products[technical metrics] and its external product and process attributes, there have actually beenvery few scientific attempts to establish specific relationships. There are a number of reasonswhy this is so; the most commonly cited is the impracticality of conducting relevantexperiments.
- Each of the "challenges" noted here is a cause for caution, but it is no reason to dismisstechnical metrics. Measurement is essential if quality is to be achieved.

Measurement Principles

- There are a series of technical metrics that:

(1) assist in the evaluation ofthe analysis and design models,

(2) provide an indication of the complexity of proceduraldesigns and source code, and

(3) facilitate the design of more effective testing,

It is important to understand basic measurement principles. Roche suggests a **measurement process** that can be characterized by **five activities**:

1. **Formulation**. The derivation of software measures and metrics that areappropriate for the representation of the software that is being considered.
2. **Collection**. The mechanism used to accumulate data required to derive theformulated metrics.
3. **Analysis**. The computation of metrics and the application of mathematical tools.
4. **Interpretation**. The evaluation of metrics results in an effort to gain insightinto the quality of the representation.
5. **Feedback**. Recommendations derived from the interpretation of technicalmetrics transmitted to the software team.

The **principles** that can be associated with the formulation of technical metrics are:

- The objectives of measurement should be established before data collectionbegins.
- Each technical metric should be defined in an unambiguous manner.
- Metrics should be derived based on a theory that is valid for the domain ofapplication (e.g., metrics for design should draw upon basic design conceptsand principles and attempt to provide an indication of the presence of anattribute that is deemed desirable).
- Metrics should be tailored to best accommodate specific products and processes.

Although formulation is a critical starting point, collection and analysis are the activitiesthat drive the **measurement process**. Roche suggests the following **principles**for these activities:

- Whenever possible, data collection and analysis should be automated.
- Valid statistical techniques should be applied to establish relationshipsbetween internal product attributes and external quality characteristics (e.g.,is the level of architectural complexity correlated with the number of defectsreported in production use?).
- Interpretative guidelines and recommendations should be established foreach metric.

In addition to these principles, the success of a metrics activity is tied to managementsupport. Funding, training, and promotion must all be considered if a technical measurementprogram is to be established and sustained.

The Attributes of Effective Software Metrics

Hundreds of metrics have been proposed for computer software, but not all providepractical support to the software engineer. Some demand measurement that is toocomplex, others are so esoteric that few real world professionals have any hopeof understanding them, and others violate the basic intuitive notions of what highqualitysoftware really is.

Ejiogudefines a set of attributes that should be encompassed by effectivesoftware metrics. The derived metric and the measures that lead to it should be:

Simple and computable. It should be relatively easy to learn how to derive themetric, and its computation should not demand inordinate effort or time.

Empirically and intuitively persuasive. The metric should satisfy the engineer'sintuitive notions about the product attribute under consideration (e.g., a metricthat measures module cohesion should increase in value as the level ofcohesion increases).

Consistent and objective. The metric should always yield results that areunambiguous. An independent third party should be able to derive the samemetric value using the same information about the software.

Consistent in its use of units and dimensions. The mathematical computationof the metric should use measures that do not lead to bizarre combinationsof units. For example, multiplying people on the project teams by programminglanguage variables in the program results in a suspicious mix of unitsthat are not intuitively persuasive.

Programming language independent. Metrics should be based on the analysismodel, the design model, or the structure of the program itself. They shouldnot be dependent on the vagaries of programming language syntax orsemantics.

An effective mechanism for high-quality feedback. That is, the metric shouldprovide a software engineer with information that can lead to a higherqualityend product.

Although most software metrics satisfy these attributes, some commonly used metricsmay fail to satisfy one or two of them. An example is the function point. It can be argued that the consistent andobjective attribute fails because an independent third party may not be able to derivethe same function point value as a colleague using the same information about thesoftware. Should we therefore reject the FP measure?

The answer is: "Of course not!"FP provides useful insight and therefore provides distinct value, even if it fails to satisfyone attribute perfectly.

3.7.Metrics for Specification Quality

Davis and his colleagues propose a list of characteristics that can be usedto assess the quality of the analysis model and the corresponding requirements specification:specificity (lack of ambiguity), completeness, correctness, understandability,verifiability, internal and external consistency, achievability, concision, traceability, modifiability,precision, and reusability.

In addition, the authors note that high-quality specificationsare electronically stored, executable or at least interpretable, annotated byrelative importance and stable, versioned, organized, cross-referenced, and specifiedat the right level of detail.

Although many of these characteristics appear to be qualitative in nature, Daviset al. suggest that each can be represented using one or more metrics. Forexample, we assume that there are nr requirements in a specification, such that

$n_r = n_f + n_{nf}$

- Wheren_f is the number of functional requirements
- n_{nf} is the number of nonfunctional(e.g., performance) requirements.

To determine the specificity (lack of ambiguity) of requirements, Davis et al. suggesta metric that is based on the consistency of the reviewers' interpretation of eachrequirement:

$Q_1 = n_{ui}/n_r$

- wheren_{ui} is the number of requirements for which all reviewers had identicalinterpretations.

  The closer the value of Q to 1, the lower is the ambiguity of thespecification.

The completeness of functional requirements can be determined by computing theratio

$Q_2 = n_u/[n_i x\ n_s]$

- where n_u is the number of unique function requirements,
- n_i is the number of inputs(stimuli) defined or implied by the specification,
- n_s is the number of states specified.

The Q2 ratio measures the percentage of necessary functions that have beenspecified for a system. However, it does not address nonfunctional requirements.

Toincorporate these into an overall metric for completeness, we must consider thedegree to which requirements have been validated:

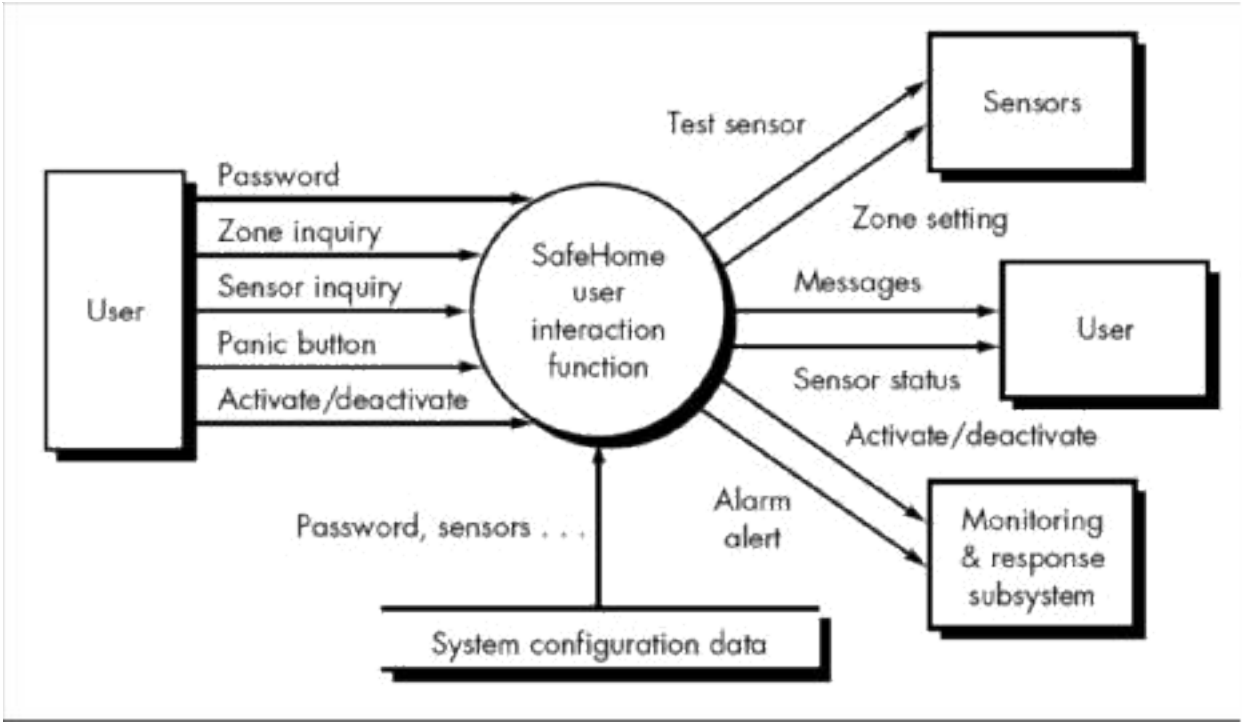
$Q_3 = n_c/[n_c + n_{nv}]$

- where n_c is the number of requirements that have been validated as correct
- $n_{nv}$is the number of requirements that have not yet been validated.

3.8.Metrics for the Analysis Model

Technical work in software engineering begins with the creation of the analysis model.

It is at this stage that requirements are derived and that a foundation for design isestablished. Therefore, technical metrics that provide insight into the quality of theanalysis model are desirable.



Although relatively few analysis and specification metrics have appeared in theliterature, it is possible to adapt metrics derived for project application for use in this context. These metrics examine the analysis model with the intent ofpredicting the "size" of the resultant system. It is likely that size and design complexitywill be directly correlated.

3.9.Metrics for the Design Model

It is inconceivable that the design of a new aircraft, a new computer chip, or a newoffice building would be conducted without defining design measures, determiningmetrics for various aspects of design quality, and using them to guide the manner inwhich the design evolves.

And yet, the design of complex software-based systemsoften proceeds with virtually no measurement. The irony of this is that design metricsfor software are available, but the vast majority of software engineers continueto be unaware of their existence.

- Design metrics for computer software, like all other software metrics, are not perfect.
- Debate continues over their efficacy and the manner in which they should beapplied. Many experts argue that further experimentation is required before designmeasures can be used. And yet, design without measurement is an unacceptablealternative.

Architectural Design Metrics

- Architectural design metrics focus on characteristics of the program architecturewith an emphasis on the architectural structure and the effectiveness ofmodules.
- These metrics are black box in the sense that they do not require any knowledgeof the inner workings of a particular software component.
- Card and Glass [CAR90] define three software design complexity measures: structuralcomplexity, data complexity, and system complexity.
- Structural complexity of a module i is defined in the following manner:

$S(i) = f^2_{out}(i)$ ---------- (19-1)

  - where$f_{out}(i)$ is the fan-out[7] of module i.

- Data complexity provides an indication of the complexity in the internal interfacefor a module i and is defined as

$D(i) = v(i)/[f_{out}(i) +1]$---------- (19-2)

  - where v(i) is the number of input and output variables that are passed to and frommodule i.

- Finally, system complexity is defined as the sum of structural and data complexity,specified as

$C(i) = S(i) + D(i)$ ---------- (19-3)

- As each of these complexity values increases, the overall architectural complexity ofthe system also increases. This leads to a greater likelihood that integration and testingeffort will also increase.
- An earlier high-level architectural design metric proposed by Henry and Kafuraalso makes use the fan-in and fan-out. The authors define a complexity metric(applicable to call and return architectures) of the form

$HKM = length(i) \times [f_{in}(i) + f_{out}(i)]^2$---------- (19-4)

  - where length(i) is the number of programming language statements in a module i
  - and fin(i) is the fan-in of a module i.

- Henry and Kafura extend the definitions of faninand fan-out presented in this book to include not only the number of module controlconnections (module calls) but also the number of data structures from which amodule i retrieves (fan-in) or updates (fan-out) data.

- To compute HKM during design,the procedural design may be used to estimate the number of programming languagestatements for module i. Like the Card and Glass metrics noted previously, an increasein the Henry-Kafura metric leads to a greater likelihood that integration and testingeffort will also increase for a module.

- Fenton suggests a number of simple morphology (i.e., shape) metrics thatenable different program architectures to be compared using a set of straightforwarddimensions. Referring to Figure 19.5, the following metrics can be defined:

Size = n + a

- where **n** is the number of nodes
- **a** is the number of arcs. For the architecture shown in Figure 19.5,
- size = 17 + 18 = 35

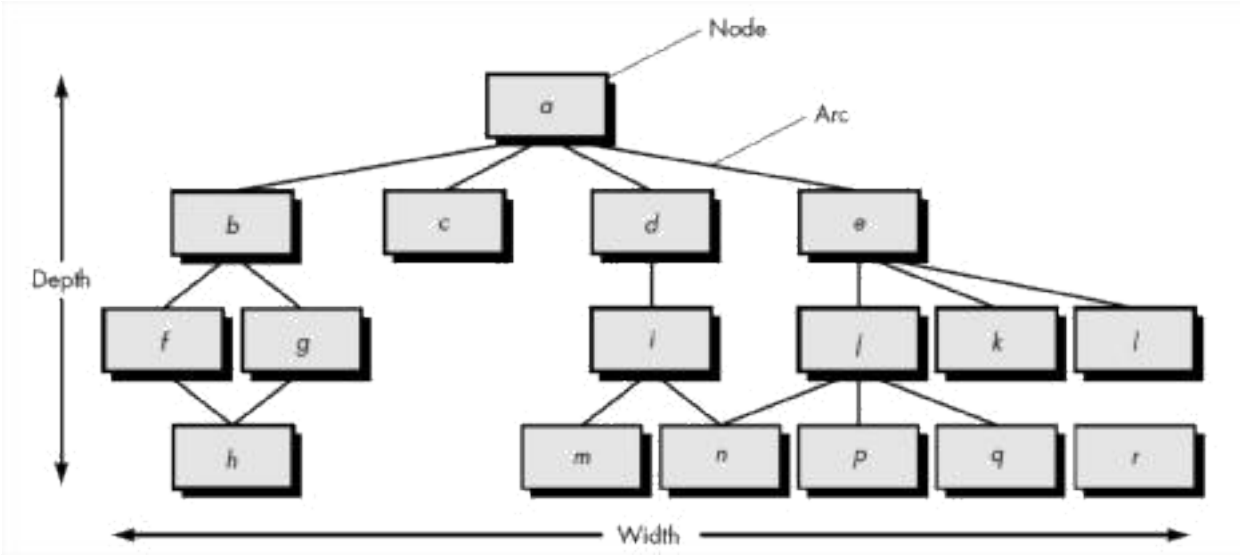


FIGURE 19.5 Morphology metrics

- **depth** = the longest path from the root (top) node to a leaf node. For the architectureshown in Figure 19.5, depth = 4.
- **width** = maximum number of nodes at any one level of the architecture. For thearchitecture shown in Figure 19.5, width = 6.

- **arc-to-node ratio**, r = a/n,which measures the connectivity density of the architecture and may provide a simpleindication of the coupling of the architecture. For the architecture shown in Figure19.5, r = 18/17 = 1.06.
- The U.S. Air Force Systems Command has developed a number of softwarequality indicators that are based on measurable design characteristics of a computerprogram. Using concepts similar to those proposed in IEEE Std. 982.1-1988
- the Air Force uses information obtained from data and architectural designto derive a **design structure quality index** (DSQI) that ranges from 0 to 1.
- The followingvalues must be ascertained to compute the DSQI:
    - S_1 = the total number of modules defined in the program architecture.
    - S_2 = the number of modules whose correct function depends on the source ofdata input or that produce data to be used elsewhere (in general, controlmodules, among others, would not be counted as part of S2).
    - S_3 = the number of modules whose correct function depends on prior processing.
    - S_4 = the number of database items (includes data objects and all attributes thatdefine objects).
    - S_5 = the total number of unique database items.
    - S_6 = the number of database segments (different records or individual objects).
    - S_7 = the number of modules with a single entry and exit (exception processingis not considered to be a multiple exit).
- Once values S_1 through S_7 are determined for a computer program, the followingintermediate values can be computed:
    - **Program structure**: D_1, where D_1 is defined as follows: If the architecturaldesign was developed using a distinct method (e.g., data flow-orienteddesign or object-oriented design), then $D_1 = 1$, otherwise $D_1 = 0$.
    - **Module independence**: $D_2 = 1 - (S_2/S_1)$
    - **Modules not dependent on prior processing**: $D3 = 1 - (S_3/S_1)$
    - **Database size**: $D_4 = 1 - (S_5/S_4)$
    - **Database compartmentalization**: $D_5 = 1 \_ (S_6/S_4)$
    - **Module entrance/exit characteristic**: $D_6 = 1 \_ (S_7/S_1)$
- With these intermediate values determined, the DSQI is computed in the followingmanner:

$DSQI = \sum W_i D_i$---------- (19-5)

- where i = 1 to 6,

- W_i is the relative weighting of the importance of each of the intermediatevalues,
- $\sum W_i = 1$ (if all D_i are weighted equally, then $W_i = 0.167$).

The value of DSQI for past designs can be determined and compared to a designthat is currently under development. If the DSQI is significantly lower than average,further design work and review are indicated. Similarly, if major changes areto be made to an existing design, the effect of those changes on DSQI can becalculated.

Component-Level Design Metrics

Component-level design metrics focus on internal characteristics of a software componentand include measures of the "three Cs"module cohesion, coupling, and complexity.
These measures can help a software engineer to judge the quality of acomponent-level design.

The metrics presented in this section are glass box in the sense that they requireknowledge of the inner working of the module under consideration. Component-leveldesign metrics may be applied once a procedural design has been developed. Alternatively,they may be delayed until source code is available.

Cohesion metrics. Bieman and Ottdefine a collection of metrics that providean indication of the cohesiveness of a module. The metrics aredefined in terms of **five** concepts and measures:

Data slice. Stated simply, a data slice is a backward walk through a modulethat looks for data values that affect the module location at which the walkbegan. It should be noted that both program slices (which focus on statementsand conditions) and data slices can be defined.

Data tokens. The variables defined for a module can be defined as datatokens for the module.

Glue tokens. This set of data tokens lies on one or more data slice.

Superglue tokens. These data tokens are common to every data slice in amodule.

Stickiness. The relative stickiness of a glue token is directly proportional tothe number of data slices that it binds.

Bieman and Ott develop metrics for **strong functional cohesion** (SFC), **weak functionalcohesion** (WFC), and **adhesiveness** (the relative degree to which glue tokens bind dataslices together). These metrics can be interpreted in the following manner:
All of these cohesion metrics range in value between 0 and 1.

They have a value of 0 whena procedure has more than one output and exhibits none of the cohesion attribute indicatedby a particular metric. A procedure with no superglue tokens, no tokens that are commonto all data slices, has zero strong functional cohesion, there are no data tokens that contributeto all outputs. A procedure with no glue tokens, that is no tokens common to morethan one data slice (in

procedures with more than one data slice), exhibits zero weak functionalcohesion and zero adhesiveness, there are no data tokens that contribute to morethan one output.

Strong functional cohesion and adhesiveness are encountered when the Bieman andOtt metrics take on a maximum value of 1.

to illustrate the character of these metrics, consider the metric forstrong functional cohesion:

SFC(i) = SG [SA(i))/(tokens(i)] ---------- (19-6)

where SG[SA(i)] denotes superglue tokens, the set of data tokens that lie on all dataslices for a module i. As the ratio of superglue tokens to the total number of tokensin a module i increases toward a maximum value of 1, the functional cohesivenessof the module also increases.

Coupling metrics. Module coupling provides an indication of the "connectedness"of a module to other modules, global data, and the outside environment.

Dhamahas proposed a metric for module coupling that encompassesdata and control flow coupling, global coupling, and environmental coupling.

Themeasures required to compute module coupling are defined in terms of each of thethree coupling types noted previously.

For data and control flow coupling:

- d_i = number of input data parameters
- c_i = number of input control parameters
- d_o = number of output data parameters
- c_o = number of output control parameters

For global coupling,

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

For environmental coupling,

- w = number of modules called (fan-out)
- r = number of modules calling the module under consideration (fan-in)

Using these measures, a module coupling indicator, mc , is defined in the following way:

- mc = k/M
- where k = 1, a proportionality constant and
- $M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r$
- where a = b = c = 2.

- The higher the value of m_c, the lower is the overall module coupling. For example,if a module has single input and output data parameters, accesses no global data,and is called by a single module,
  - $m_c = 1/(1 + 0 + 1+ 0 + 0 + + 0 + 1 + 0) = 1/3 = 0.33$
- We would expect that such a module exhibits low coupling. Hence, a value of mc =0.33 implies low coupling. Alternatively, if a module has five input and five outputdata parameters, an equal number of control parameters, accesses ten items of globaldata, has a fan-in of 3 and a fan-out of 4,and the implied coupling would be high.
  - $m_c = 1/[5 + (2 \times 5) + 5 + (2 \times 5) + 10 + 0 + 3 + 4] = 0.02$
- In order to have the coupling metric move upward as the degree of coupling increases, a revised coupling metricmay be defined as
  - $C = 1 - m_c$
- Where the degree of coupling increases nonlinearly between a minimum values in therange 0.66 to a maximum value that approaches 1.0.
- Complexity metrics. A variety of software metrics can be computed to determinethe complexity of program control flow. Many of these are based on the flow graph.
- A graph is a representation composed of nodes andlinks (also called edges). When the links (edges) are directed, the flow graph is a directedgraph.
- McCabe and Watson identify a number of important uses for complexitymetrics:
- Complexity metrics can be used to predict critical information about reliability andmaintainability of software systems from automatic analysis of source code [or proceduraldesign information]. Complexity metrics also provide feedback during the softwareproject to help control the [design activity].
- During testing and maintenance, theyprovide detailed information about software modules to help pinpoint areas of potentialinstability.
- The most widely used (and debated) complexity metric for computer software iscyclomatic complexity, originally developed by Thomas McCabe.
- The McCabe metric provides a quantitative measure of testing difficulty and anindication of ultimate reliability. Experimental studies indicate distinct relationshipsbetween the McCabe metric and the number of errors existing in source code, as wellas time required to find and correct such errors.
- McCabe also contends that cyclomatic complexity may be used to provide a quantitativeindication of maximum module size. Collecting data from a number of actualprogramming projects, he has found that cyclomatic complexity = 10 appears to bea practical upper limit for module size.
- When the cyclomatic complexity of modulesexceeded 10, it became extremely difficult to adequately test a module.

Zuse presents an encyclopedic discussion of no fewer that 18different categories of software complexity metrics. The author presents the basicdefinitions for metrics in each category and then analyzes and critiques each. Zuse's work isthe most comprehensive published to date.

Interface Design Metrics

- Although there is significant literature on the design of human/computer interfaces, relatively little information has been published on metrics that wouldprovide insight into the quality and usability of the interface.
- Sears suggests that layout appropriateness (LA) is a worthwhile designmetric for human/computer interfaces. A typical GUI uses layout entities, graphicicons, text, menus, windows, and the like, to assist the user in completing tasks.
- Toaccomplish a given task using a GUI, the user must move from one layout entity tothe next. The absolute and relative position of each layout entity, the frequency withwhich it is used, and the "cost" of the transition from one layout entity to the next allcontribute to the appropriateness of the interface.
- For a specific layout (i.e., a specific GUI design), cost can be assigned to eachsequence of actions according to the following relationship:
  - cost = ∑ [frequency of transition(k) x cost of transition(k)] ---------- (19-7)
- Where k is a specific transition from one layout entity to the next as a specific task isaccomplished.
- The summation occurs across all transitions for a particular task orset of tasks required to accomplish some application function.
- Cost may be characterizedin terms of time, processing delay, or any other reasonable value, such as thedistance that a mouse must travel between layout entities. Layout appropriatenessis defined as:
  - LA = 100 x [(cost of LA - optimal layout)/(cost of proposed layout)] ------ (19-8)
  - where LA = 100 for an optimal layout.
- To compute the optimal layout for a GUI, interface real estate (the area of thescreen) is divided into a grid. Each square of the grid represents a possible positionfor a layout entity. For a grid with N possible positions and K different layout entitiesto place, the number of possible layouts is represented in the following manner:
  - number of possible layouts = [N!/(K! x (N - K)!] x K! ------(19-9)
- As the number of layout positions increases, the number of possible layouts growsvery large.
- To find the optimal (lowest cost) layout, Sears proposes a treesearching algorithm.
- LA is used to assess different proposed GUI layouts and the sensitivity of a particularlayout to changes in task descriptions (i.e., changes in the sequence and/or frequencyof transitions). The interface designer

can use the change in layout appropriateness,ΔLA, as a guide in choosing the best GUI layout for a particular application.

- It is important to note that the selection of a GUI design can be guided with metricssuch as LA, but the final arbiter should be user input based on GUI prototypes.
- Nielsen and Levy report that "one has a reasonably large chance of success if one chooses between interface [designs] based solely on users' opinions. Users'average task performance and their subjective satisfaction with a GUI are highly correlated."

3.10.Metrics for Source Code

- Halstead's theory of software science is one of "the best known and mostthoroughly studied . . . composite measures of (software) complexity".
- Softwarescience proposed the first analytical "laws" for computer software.
- Software science assigns quantitative laws to the development of computer software,using a set of primitive measures that may be derived after code is generatedor estimated once design is complete. These follow:
  - n_1 = the number of distinct operators that appear in a program.
  - n_2 = the number of distinct operands that appear in a program.
  - N_1 = the total number of operator occurrences.
  - N_2 = the total number of operand occurrences.
- Halstead uses these primitive measures to develop expressions for the overall programlength, potential minimum volume for an algorithm, the actual volume (numberof bits required to specify a program), the program level (a measure of software complexity),the language level (a constant for a given language), and other features suchas development effort, development time, and even the projected number of faultsin the software.
- Halstead shows that length N can be estimated
  - $N = n1 \log_2 n_1 + n_2 \log_2 n_2$---------- (19-10)

and program volume may be defined

  - $V = N \log_2 (n_1 + n_2)$ ----------(19-11)
- It should be noted that V will vary with programming language and represents thevolume of information (in bits) required to specify a program.
- Theoretically, a minimum volume must exist for a particular algorithm. Halsteaddefines a volume ratio L as the ratio of volume of the most compact form of a programto the volume of the actual program. In actuality, L must always be less than 1.

- In terms of primitive measures, the volume ratio may be expressed as: o $L = 2/n_1 x\ n_2/N_2$----------(19-12)
- Halstead's work is amenable to experimental verification and a large body ofresearch has been conducted to investigate software science.
- good agreement has beenfound between analytically predicted and experimental results.

3.11.Metrics for Testing

- Although much has been written on software metrics for testing, themajority of metrics proposed focus on the process of testing, not the technical characteristicsof the tests themselves.
- In general, testers must rely on analysis, design,and code metrics to guide them in the design and execution of test cases.
- Function-based metrics can be used as a predictor for overall testingeffort. Various project-level characteristics (e.g., testing effort and time, errorsuncovered, number of test cases produced) for past projects can be collected and correlatedwith the number of FP produced by a project team.
- The team can then project"expected values" of these characteristics for the current project.
- The bang metric can provide an indication of the number of test cases required byexamining the primitive measures. The number of functionalprimitives (FuP), data elements (DE), objects (OB), relationships (RE), states(ST), and transitions (TR) can be used to project the number and types of black-boxand white-box tests for the software.
- For example, the number of tests associatedwith the human/computer interface can be estimated by:

  (1) examining the numberof transitions (TR) contained in the state transition representation of the HCI and evaluatingthe tests required to exercise each transition;

  (2) examining the number ofdata objects (OB) that move across the interface, and

  (3) the number of data elementsthat are input or output.

- Architectural design metrics provide information on the ease or difficulty associated with integration testing and the need for specialized testing software (e.g.,stubs and drivers).
- Cyclomatic complexity (a component-level design metric) lies atthe core of basis path testing,
- Inaddition, cyclomatic complexity can be used to target modules as candidates for extensive unit testing. Modules with high cyclomatic complexity are more likelyto be error prone than modules whose cyclomatic complexity is lower. For this reason,the tester should expend above average effort to uncover errors in such modulesbefore they are integrated in a system.
- Testing effort can also be estimated using metrics derived from Halstead measures. Using the definitions for programvolume, V, and program level, PL, software science effort, e, can be computed as:

- PL = $1/[(n_1/2)x(N_2/n_2)]$---------- (19-13a)
- e = V/PL ---------- (19-13b)

The percentage of overall testing effort to be allocated to a module k can be estimatedusing the following relationship:

- percentage of testing effort (k) = e(k)/ ∑ e(i)---------- (19-14)
- where e(k) is computed for module k using Equations (19-13) and the summation inthe denominator of Equation (19-14) is the sum of software science effort across allmodules of the system.

As tests are conducted, three different measures provide an indication of testingcompleteness. A measure of the breath of testing provides an indication of how manyrequirements (of the total number of requirements) have been tested.

This providesan indication of the completeness of the test plan. Depth of testing is a measure ofthe percentage of independent basis paths covered by testing versus the total numberof basis paths in the program.

A reasonably accurate estimate of the number ofbasis paths can be computed by adding the cyclomatic complexity of all programmodules.

Finally, as tests are conducted and error data are collected, fault profilesmay be used to rank and categorize errors uncovered. Priority indicates the severityof the problem. Fault categories provide a description of an error so that statisticalerror analysis can be conducted.

3.12.Metrics for Maintenance

All of the software metrics introduced in this chapter can be used for the developmentof new software and the maintenance of existing software. However, metricsdesigned explicitly for maintenance activities have been proposed.

IEEE Std. 982.1-1988 suggests a **softwarematurityindex** (SMI) that providesan indication of the stability of a software product (based on changes that occur foreach release of the product). The following information is determined:

- M_T = the number of modules in the current release
- F_c = the number of modules in the current release that have been changed
- F_a = the number of modules in the current release that have been added
- F_d = the number of modules from the preceding release that were deleted inthe current release

The software maturity index is computed in the following manner:

- SMI = $[M_T - (F_a + F_c + F_d)]/M_T$---------- (19-15)

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as metricfor planning software maintenance activities. The mean time to produce a releaseof a software product can be correlated with SMI and empirical models for maintenanceeffort can be developed.

3.13.Technical Metrics For Object-Oriented Systems:

3.13.1.Intent of object oriented metrics:

Goals for Using Object-Oriented Metrics

• To better understand product quality

• To assess process effectiveness

• To improve quality of the work performed at the project level

3.13.2.Characteristics of Object-Oriented Metrics

• Localization - OO metrics need to apply to the class as a whole and should reflect the manner in which classes collaborate with one another

• Encapsulation - OO metrics chosen need to reflect the fact that class responsibilities, attributes, and operations are bound as a single unit

• Information hiding -OO metrics should provide an indication of the degree to which information hiding has been achieved

• Inheritance - OO metrics should reflect the degree to which reuse of existing classes has been achieved

• Abstraction - OO metrics represent abstractions in terms of measures of a class (e.g. number of instances per class per application)

3.13.3.Object-Oriented Design Model Metrics

• Size(population, volume, length, functionality)

• Complexity (how classes interrelate to one another)

• Coupling (physical connections between design elements)

• Sufficiency (how well design components reflect all properties of the problem domain)

• Completeness (coverage of all parts of problem domain)

• Cohesion (manner in which all operations work together)

• Primitiveness (degree to which attributes and operations are atomic)

• Similarity (degree to which two or more classes are alike)

• Volatility (likelihood a design component will change)

3.13.4.Class-Oriented Metrics

• Chidamber and Kemerer (CK) Metrics Suite

• weighted metrics per class (WMC)

• depth of inheritance tree (DIT)

• number of children (NOC)

• coupling between object classes (CBO)

• response for a class(RFC)

• lack of cohesion in methods (LCOM)

• Lorenz and Kidd • class size (CS)

• number of operations overridden by a subclass (NOO)

• number of operations added by a subclass (NOA)

• specialization index (SI)

• Harrison, Counsel, and Nithi (MOOD) Metrics Suite

• method inheritance factor (MIF)

• coupling factor (CF)

• polymorphism factor (PF)

3.13.5.Operation-Oriented Metrics

• Average operation size (OSavg)

• Operation complexity (OC)

• Average number of parameters per operation (NPavg)

3.13.6.Object-Oriented Testing Metrics

• Encapsulation

• lack of cohesion in methods (LCOM)

• percent public and protected (PAP)

• public access to data members(PAD)

• Inheritance

• number of root classes (NOR)

• fan in (FIN)

• number of children (NOC)

• depth of inheritance tree (DIT)

• Class complexity

• weighted metrics per class(WMC)

• coupling between object classes (CBO)

• response for a class (RFC)

3.13.7.Metrics for object-oriented projects:

A software team can use software project metrics to adapt project workflow and technical activities.

Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.

Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).