CSE1005: Software Engineering

Module No. 4: Process and Product Metrics

Dr. K Srinivasa Reddy
SCOPE, VIT-AP University

Module No. 4: Process and Product Metrics

Product Metrics, Metrics for the Requirements Model, Metrics for the Design Model, Architectural Design Metrics, Metrics for Software Quality

Text Book:

1. Roger Pressman, "Software Engineering: A Practitioner's Approach", McGraw-Hill, 7th Edition, 2016

Course Outcome: Apply quality assurance techniques at the module level, and understand these techniques at the system, organization level (CO4)

Metrics - Introduction

Examples of Metrics from Everyday Life

College

- Grades received in last semester
- Number of classes attended each semester
- Amount of time spent in class this week
- Amount of time spent on studying and homework this week
- Number of hours of sleep last night

Working and living

- Cost of utilities for the month
- Cost of groceries for the month
- Amount of rent per month
- Time spent at work each Saturday for the past month
- Time spent cutting the lawn for the past two times

Travel

- Time to drive from home to the college
- Amount of KMs traveled today
- Cost of meals and lodging for yesterday

Metrics - Introduction

- **Quantitative** and **Qualitative** methods generate different types of data.
  - **Quantitative** data is expressed as **numbers**
  - **Qualitative** data is expressed as **words**
- By its nature, **engineering is a quantitative discipline**.
  - Unfortunately, unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics.
- Direct measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world.
- Because software measures and metrics are often indirect, they are open to debate.

Product Metrics

Why to have Software Product Metrics?

Help software engineers to

- **Better understand** the **attributes of models** and **assess the quality** of the software **based on a set of clearly defined rules**
- **Gain insight** into the **design and construction** of the software
- **Focus on specific attributes** of software engineering **work products** resulting from analysis, design, coding, and testing
- Provide an **“on-the-spot”** rather than **“after-the-fact”** insight into the software development

Product Metrics

Outline of Topics:

- A Framework for Product Metrics
- Metrics for the Requirements Model
- Metrics for the Design Model
- Architectural Design Metrics
- Metrics for Software Quality

A Framework for Product Metrics

Measures, Metrics, and Indicators:

- How old are you?
- How much do you weigh?
- How tall are you?
- How much water can be filled in water bottle?
- How hot is it today?
- To answer the above questions;

  To know - Need to measure

  - How old you are - **time**.
  - How much you weigh - **weigh yourself.**
  - How tall you are - **height (length)**
  - How much water can be filled in water bottle - **the capacity of bottle.**
  - How hot it is today - **the temperature**.

- Measurement is the **process of finding a number** that shows the amount of something.

A Framework for Product Metrics
Measures, Metrics, and Indicators:

- These three terms are often used interchangeably, but they can have subtle differences
  - **Measure**
    - Provides a **quantitative sign** of the magnitude, amount, dimension, capacity, or size of some attribute of a product or process
    - **Measurement**
      - The act of determining a measure
  - **Metric**
    - A **quantitative measure** of the degree to which a system, component, or process **possesses** a given attribute.
    - [standard way of measuring distance, calculating height, and most of the other day-to-day items – liters, meters, seconds, kgs etc..]

A Framework for Product Metrics

Measures, Metrics, and Indicators:

- These three terms are often used interchangeably, but they can have subtle differences
  - **Measure** :Provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
    - **Measurement** :The act of determining a measure
    - Example: Number of errors
  - **Metric** :A quantitative measure of the degree to which a system, component, or process possesses a given attribute.
    - Example: Number of errors found per person hours expended
  - **Indicator**
    - A **metric or combination of metrics** that provides insight into the software process, a software project, or the product itself.
    - A software engineer **collects** measures and **develops** metrics so that indicators will be obtained

A Framework for Product Metrics

Measures, Metrics, and Indicators:

Example:

- When a **single data point has been collected** (e.g., the number of errors uncovered within a single software component), a **measure** has been established
- **Measurement** occurs as the result of the **collection of one or more data points** (e.g., a number of component reviews and unit tests are investigated to collect measures of the number of errors for each).
- A **software metric** relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test)
- An **indicator** provides insight that enables the project manager or software engineers to adjust the process, the project, or the product to make things better.

A Framework for Product Metrics

Purpose of Product Metrics

- Helps in the evaluation of analysis and design models
- Provide an indication of the complexity of procedural designs and source code
- Facilitate the design of more effective testing techniques

A Framework for Product Metrics

Activities of a Measurement Process:

- **Formulation**
    - The **derivation (i.e., identification)** 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** in an effort **to gain insight into the quality of the representation**
- **Feedback**
    - **Recommendations derived** from the interpretation of product metrics and **passed on to the software development team**.

A Framework for Product Metrics

Characterizing and Validating Metrics:

- Software metrics will be **useful** only if they are characterized effectively and validated so that their worth is proven.
- A **metric should have desirable mathematical properties**
  - It should have a meaningful range (e.g., zero to ten)
  - It should **not be set on a rational scale (ratios between values are meaningful)** if it is composed of components measured on **an ordinal scale (no information about the distance between the values is given)** like

1- Very Unhappy,	1- Totally Satisfied,
2- Unhappy,	2- Satisfied,
3- Neutral,	3- Neutral,
4- Unhappy,	4- Dissatisfied,
5- Very Unhappy	5- Totally Dissatisfied,
	not specify how much

A Framework for Product Metrics

Characterizing and Validating Metrics:

- When a metric represents a software characteristic that **increases when positive traits occur or decreases when undesirable traits are encountered**, the value of the metric **should increase or decrease in the same manner**.

- Each metric **should be validated empirically** in a wide variety of contexts before being published or used to make decisions
  - It should **measure the factor of interest independently** of other factors
  - It should **scale up to large systems**
  - It should **work in a variety** of programming languages and system domains

A Framework for Product Metrics

Collection and Analysis Guidelines

- Whenever possible, **data collection and analysis should be automated**
- Valid **statistical techniques** should be applied **to establish relationships** between **internal product attributes** and **external quality characteristics**
- Interpretative **guidelines and recommendations** should be established for each metric.

A Framework for Product Metrics

The Attributes of Effective Software Metrics

- Simple and computable
- Consistent and objective (unambiguous)
- Use consistent units and dimensions
- Programming language independent
- Easy and Cost effective to obtain
- An effective mechanism for high-quality feedback

Product Metrics

Outline of Topics:

- A Framework for Product Metrics
- Metrics for the Requirements Model
- Metrics for the Design Model
- Architectural Design Metrics
- Metrics for Software Quality

A Framework for Product Metrics
Measures, Metrics, and Indicators:

- These three terms are often used interchangeably, but they can have subtle differences
  - **Measure** :Provides a **quantitative indication** of the extent, amount, dimension, capacity, or size of some attribute of a product or process
    - **Measurement** :The act of determining a measure
  - **Metric** :A **quantitative measure of the degree** to which a system, component, or process possesses a given attribute.
  - **Indicator**
    - A **metric or combination of metrics** that **provides insight into the software process**, a project, or the product itself.
    - A software engineer **collects** measures and **develops** metrics so that indicators will be obtained

Metrics for the Requirements Model

- **Technical work** in software engineering **begins with the creation of the requirements model.**
- **Examine** the requirements model with the intent of **predicting the “size” of the resultant system.**
- **Size** is an indicator of **design complexity** and **coding, integration, and testing effort.**
- **Metrics for the Requirements Model**
    - Function-Based Metrics
    - Metrics for Specification Quality

Metrics for the Requirements Model

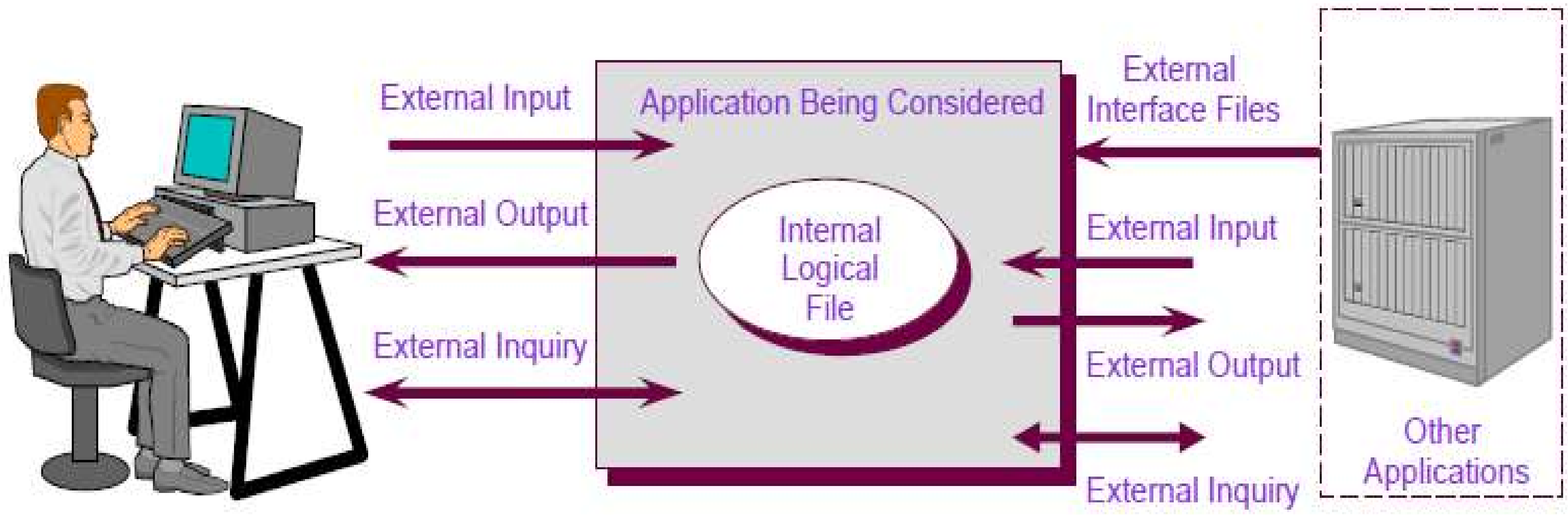
Function-Based Metrics (Function Point (FP) metric):

- **Used** effectively as a means for **measuring the functionality delivered** by the system.
- It measures the logical view of an application, not the physically implemented view or the internal technical view.
- Using historical data, the FP metric can be used to
  1. **Estimate the cost or effort** required **to design, code, and test the software**
  2. **Predict the number of errors** that will be **encountered during testing**; and
  3. **Estimate the number of components** and/or the **number of projected source lines** in the implemented system.
- Function Points are **derived** using a **relationship between the complexity of software** and **the information domain value.**

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

- **Information Domain Values (IDV)** used in function point include



Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

- **Information Domain Values (IDV)** used in function point include

Measurements Parameters	Examples
1. Number of External Inputs (EI)	Input screen and tables
2. Number of External Outputs (EO)	Output screens and reports
3. Number of External inquiries (EQ)	Prompts and interrupts, data sent out of application without added value
4. Number of internal files (ILF)	Databases and directories
5. Number of external interfaces (EIF)	Shared databases and shared routines.

- All these parameters are **individually assessed for complexity.**
- The IDV's categorized into 2 types:
  - **Transactional** Functional type : EI, EO, EQ
  - **Data** Functional type: ILF, EIF

Metrics for the Requirements Model

Number of external inputs (EIs):

- Originates from a **user** or **transmitted from another application** and provides distinct application-oriented data or control information.
- Inputs are used to update *internal logical files* (ILFs).
- Inputs should be distinguished from inquiries, which are counted separately.

Number of external outputs (EOs):

- **Derived data within the application** that provides information to the user.
- Refers to **reports, screens, error messages**, etc.
- Individual data items within a report are not counted separately.

Number of external inquiries (EQs):

- An **online input** that results in the **generation of some immediate** software **response in the form of an online output** (often retrieved from an ILF).

Number of internal logical files (ILFs):

- A logical grouping of data that resides **within the application's boundary** and is **maintained via external inputs**.

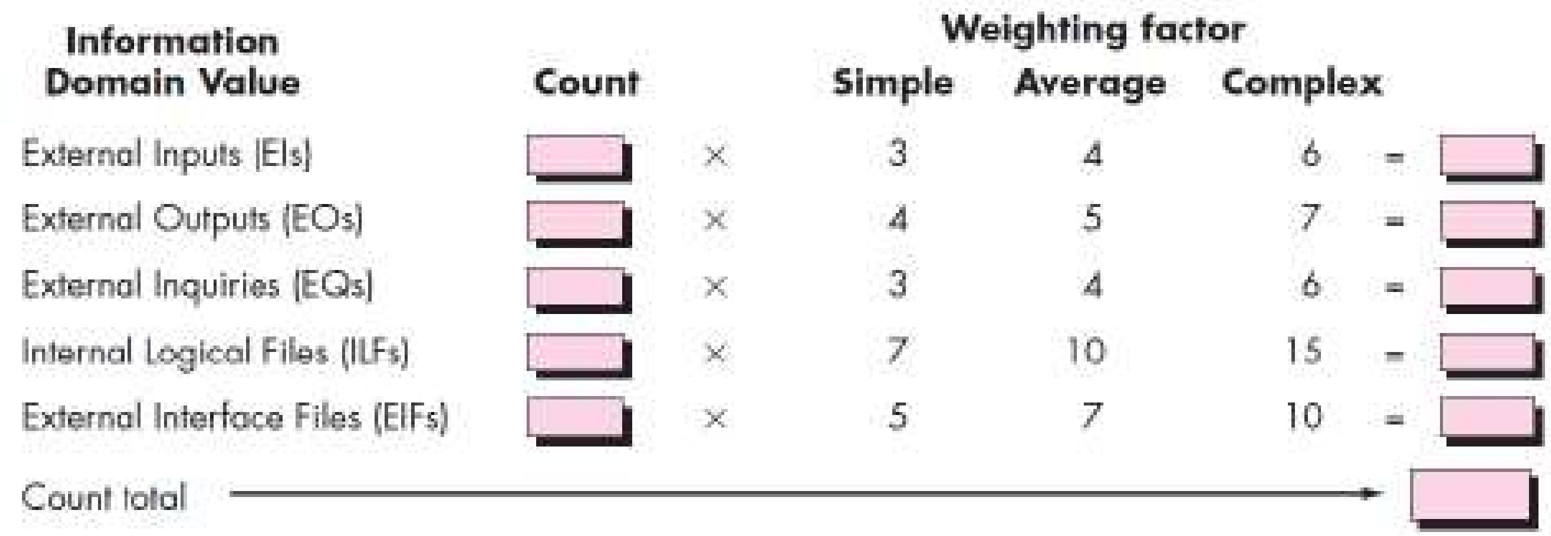
Number of external interface files (EIFs):

- A logical grouping of data that resides **external to the application** but provides information that may be of use to the application.

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

- Once the data have been collected, Calculate the Count total (or) **UFP (Unadjusted Function Point)** by using the Complexity Value table with a complexity value associated with each count.
- Organizations that use function point methods develop criteria for determining whether a particular entry is **simple, average, or complex.**



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

- The functional complexities are **multiplied** with the corresponding weights against each function, and the values are added up to determine the **UFP** of the subsystem.

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

- The Function Point (FP) formula **FP = Count total * [0.65 + 0.01 * $\sum(F_i)$]**

  Constants (0.65 and 0.01) derived using historical data
- The F_i (*i* =1 to 14) are ***Value Adjustment Factors*** **(VAF)** or **Complexity Adjustment Values (CAV)** **based on responses to the following questions:**

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require online data entry?
7. Does the online data entry require the input transaction to be built over multiple screens or operations?

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

- The Function Point (FP) formula **FP = Count total * [0.65 + 0.01 * ∑(F_i)]**
- The F_i (i =1 to 14) are ***Value Adjustment Factors*** **(VAF) or Complexity Adjustment Values (CAV)** based on responses to the questions.

8. Are the ILFs updated online?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

The above questions are answered on a scale that ranges from **0 - No Influence, 1 - Incidental, 2 - Moderate, 3 - Average, 4 - Significant, 5 - Essential**

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

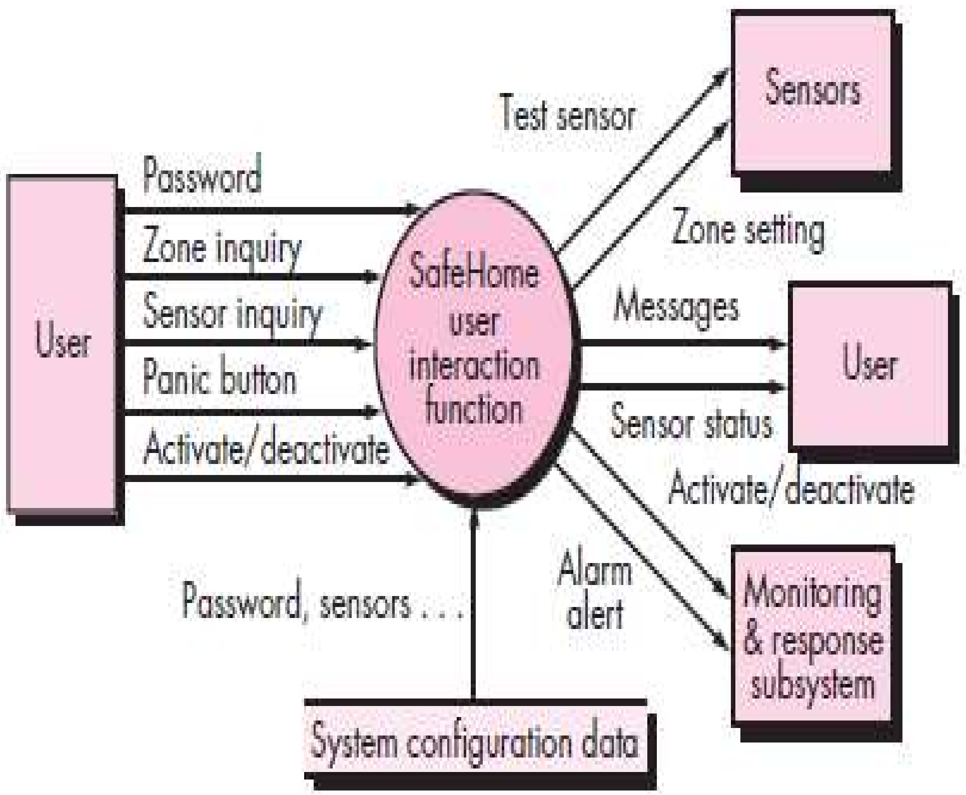
- The Function Point (FP) formula

  FP = Count total * [0.65 + 0.01 * $\sum(F_i)$]

- The F_i (i =1 to 14) are ***Value Adjustment Factors*** **(VAF) or Complexity Adjustment Values (CAV)** based on responses to the questions.
- Usually, **the value of $\sum(F_i)$ is provided as 0 (no influence, not important or not applicable), 3 (average), 5 (absolutely essential).**
- $\sum(F_i)$ ranges from 0 to 70
  - If $\sum(F_i)$ = 0, **FP = Count total * 0.65**
  - If $\sum(F_i)$ = 42, **FP = Count total * (0.65 + 0.01 * 42) = Count total * 1.07**
  - If $\sum(F_i)$ = 70, **FP = Count total * (0.65 + 0.01 * 70) = Count total * 1.35**

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):



A data flow model for *SafeHome* software

The function manages

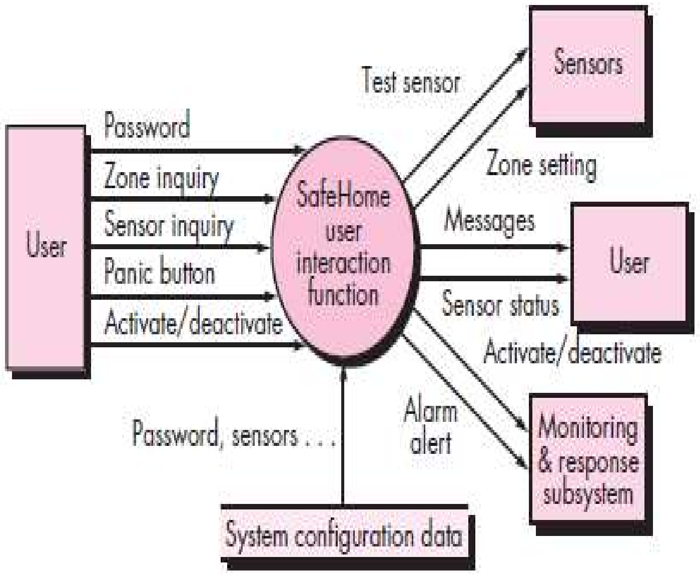
- **User interactions** like
  - Accepting a user password to activate or deactivate the system, and
  - allows inquiries on the status of security zones and various security sensors.
- The function **displays** a series of prompting messages and **sends appropriate control signals** to various components of the security system.

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

Number of

- External Inputs(EI): 03 - **Password, panic button**, and **activate/deactivate**
- External Output (EO): 02 - **Messages** and **sensor status**
- External inquiries (EQ): 02 - **Zone inquiry** and **sensor inquiry**
- Internal files (ILF) : 01 - **System configuration file**
- External interfaces (EIF) : 04 - **Test sensor, zone setting, activate/deactivate,** and **alarm alert**

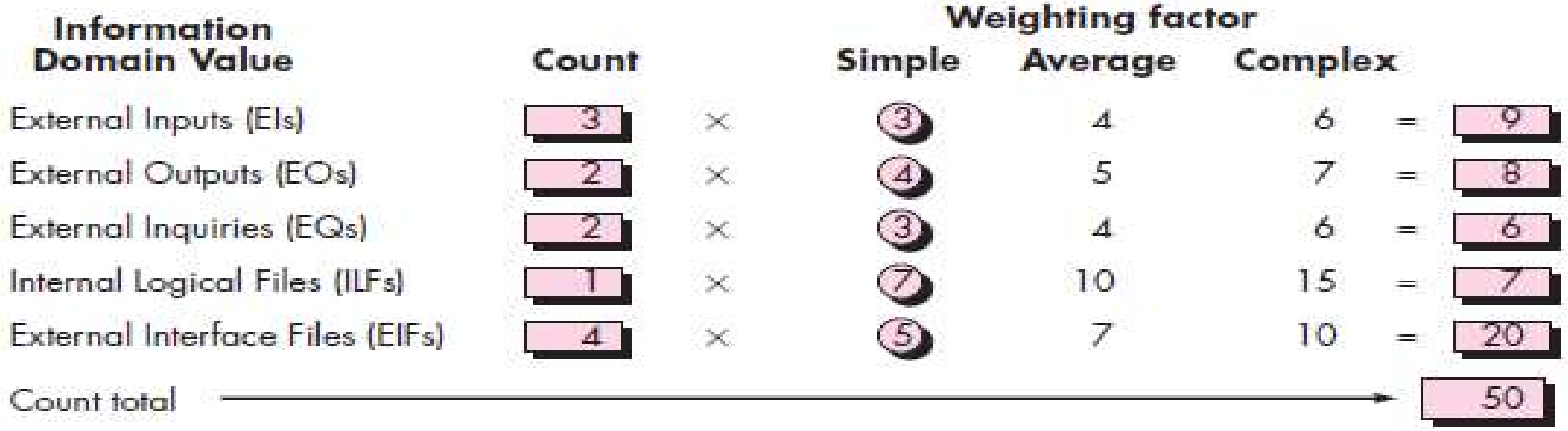


A data flow model for *SafeHome* software

Metrics for the Requirements Model

Function-Based Metrics (Function Point (FP) metric):

Information domain value data, along with the appropriate complexity (simple) :



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

- The Function Point (FP) formula

$$FP = \text{Count total} * [0.65 + 0.01 * \sum(F_i)]$$

Let us assume $\sum(F_i) = 42$ [All factors are average, 14 * 3 = 42]

$$FP = \text{Count total} * [0.65 + 0.01 * \sum(F_i)]$$

$$= 50 * [0.65 + 0.01 * 42] = 53.5$$

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

Metrics for the Requirements Model

Example:

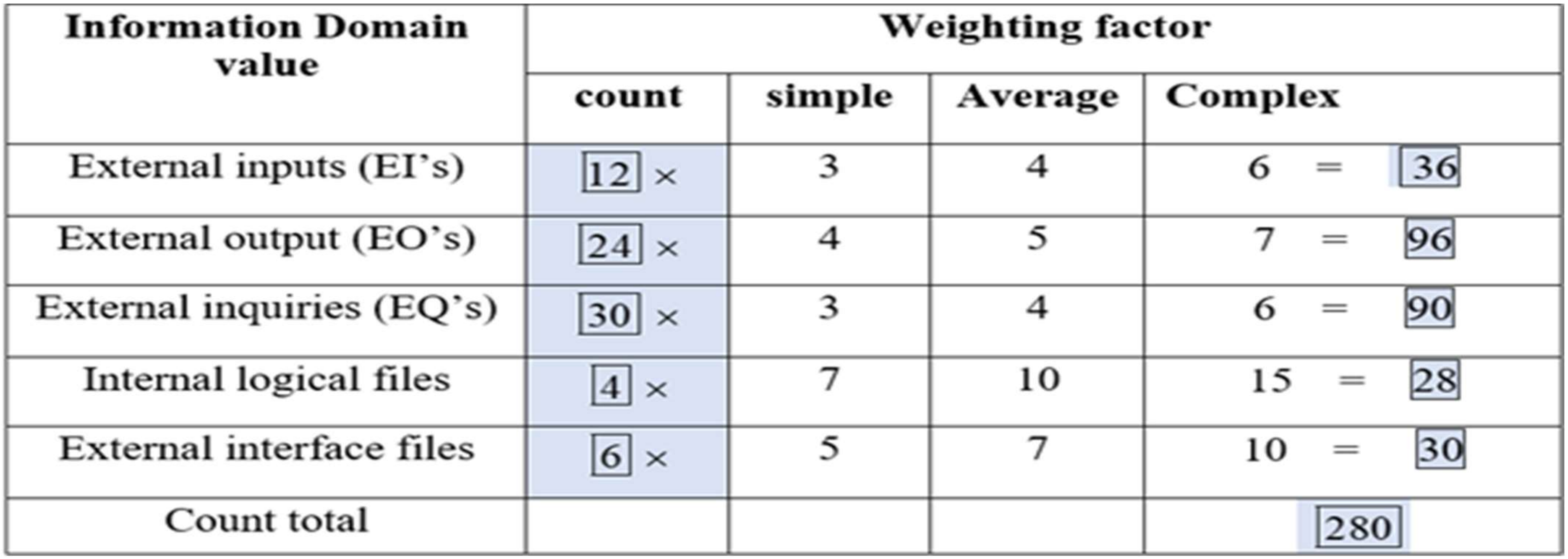
- A system has 12 external inputs, 24 external outputs, fields 30 different external queries, manages 4 internal logical files and interfaces with 6 different legacy system (6 EIFs).
- All of these data are of average complexity, and the overall system is relatively simple. Compute FP for the system.

- How to identify simple, average or complex and $\sum$**(F$_i$)**
  - All of these data are of **average complexity**, and the **overall system is relatively simple**.

- All of these data are of average complexity = 3. $\sum$**(F$_i$) = 14 * 3 = 42**
- **Overall system is relatively <u>simple</u>.**

Metrics for the Requirements Model

- How to identify simple, average or complex and $\sum(F_i)$
- All of these data are of **average complexity**, and the **overall system is relatively simple**. Compute FP for the system.
- All of these data are of average complexity = 3. $\sum(F_i)$ **= 14 * 3 = 42**



Information Domain value	Weighting factor				
	count	simple	Average	Complex	
External inputs (EI's)	$\boxed{12}$ ×	3	4	6 =	$\boxed{36}$
External output (EO's)	$\boxed{24}$ ×	4	5	7 =	$\boxed{96}$
External inquiries (EQ's)	$\boxed{30}$ ×	3	4	6 =	$\boxed{90}$
Internal logical files	$\boxed{4}$ ×	7	10	15 =	$\boxed{28}$
External interface files	$\boxed{6}$ ×	5	7	10 =	$\boxed{30}$
Count total					$\boxed{280}$

FP = Count total * (0.65 + 0.01 * 42) = 280 * (0.65 + 0.42) = 299.6

Metrics for the Requirements Model

Exercise: Compute the function point, productivity, documentation, cost per function for the following data: Number of user inputs = 24, Number of user outputs = 46, Number of inquiries = 8, Number of files = 4, Number of external interfaces = 2. Various processing complexity factors are: 4, 1, 0, 3, 3, 5, 4, 4, 3, 3, 2, 2, 4, 5. Assume that weights are average for user inputs, internal files - average, inquiries – complex, outputs, external interfaces – simple.

Measurement Parameter	**Count**		**Weighing factor**
1. Number of external inputs (EI)	24	*	4 = 96
2. Number of external outputs (EO)	46	*	4 = 184
3. Number of external inquiries (EQ)	8	*	6 = 48
4. Number of internal files (ILF)	4	*	10 = 40
5. Number of external interfaces (EIF)	2	*	5 = 10
Count total			378

$\sum(F_i) = 4 + 1 + 0 + 3 + 5 + 4 + 4 + 3 + 3 + 2 + 2 + 4 + 5 = 43$

FP = Count total * (0.65 + 0.01 * 43) = 378 * (1.08) = 408

Metrics for the Requirements Model

Exercise: Compute the function point value for a project with the following information domain characteristics: Number of user inputs: 32 Number of user outputs: 60 Number of user enquiries: 24 Number of files: 8 Number of external interfaces: 2. Assume that weights are average and external complexity adjustment values are not important.

- How to **identify simple, average or complex** and $\sum(F_i)$
- Weights are **average**
- **Complexity adjustment values are not important, $\sum(F_i) = 14 * 0 = 0$**

FP = Count total * (0.65 + 0.01 * 0)
= 618 * (0.65) = 401.7

Metrics for the Requirements Model

Exercise: Compute the function point value for a project with the following information domain characteristics: Number of user inputs: 32 Number of user outputs: 60 Number of user enquiries: 24 Number of files: 8 Number of external interfaces: 2. Assume that weights are average and external complexity adjustment values are average.

- How to **identify simple, average or complex** and **$\sum(F_i)$ =**
- Weights are **average**
- **Complexity adjustment values are not important, $\sum(F_i)$ = 14 * 3 = 42**

FP = Count total * (0.65 + 0.01 * 42)
= 618 * (1.07) = 661.26

Metrics for the Requirements Model

Metrics computed based on the Function Point Metric:

- **Productivitity = FP / Effort (effort is measured in person-months)**
- **Documentation = Pages of documentation / FP**
- **Pages of documentation = Technical document + User document**
- **Cost per Function = Cost / Productivitity**

Metrics for the Requirements Model

Let us say FP = 408, Effort = 36.9 p-m, Technical documents = 265 pages User documents = 122 pages, and Cost = $7744/ month
Find Productivitity, Total pages of documentation, Documentation, Cost per Function

- **Productivitity = FP / Effort**
  = 408 / 36.9 = 11.1
- **Total pages of documentation = Technical document + User document**
  = 265 + 122 = 387 Pages
- **Documentation = Pages of documentation / FP**
  = 387 / 408 = 0.94
- **Cost per Function = Cost / Productivitity**
  = 7744 / 11.1 = $700

Metrics for the Requirements Model

Metrics for Specification Quality:

- Set of characteristics has been proposed **to evaluate the quality of analysis model and requirements specification:**
  - Specificity,
  - Completeness,
  - Correctness,
  - Understandability,
  - Verifiability,
  - Internal and External consistency,
  - Achievability,
  - Concision,
  - Traceability,
  - Modifiability,
  - Precision, and
  - Reusability

- Many of these characteristics are **qualitative** in nature.
- However, each can be represented using one or more metrics.

Metrics for the Requirements Model

Metrics for Specification Quality:

Given n_f *and* n_{nf}, where

n_f = The number of **functional requirements** and

n_{nf} = The number of **nonfunctional requirements**

Requirements in a specification $\boldsymbol{n_r = n_f + n_{nf}}$

Metric for Specificity of requirements: A metric based on the **consistency of the reviewer's understanding** of each requirement has been proposed.

$$\boldsymbol{Q_1 = n_{ui} / n_r}$$

where

- n_{ui} = The no. of requirements for which all reviewers have similar understanding.
- Q_1 = Specificity
- The closer the value of Q to 1, the lower is the ambiguity of the specification.

Metrics for the Requirements Model

Metric for Completeness of requirements:

$$Q_2 = n_u / (n_i * n_s)$$

where

- n_u = The no. of unique functional requirements.
- n_i = The no. of inputs defined by the specification
- n_s = The no. of states specified

- Q_2 ratio **considers only functional requirements** and **ignores non-functional requirements.**
- In order to consider **non-functional requirements**, it is necessary to **consider the degree** to which **requirements have been validated**.

$$Q_3 = n_c / (n_c + n_{nv})$$

- n_c = The number of requirements **validated as correct** and
- n_{nv} = The number of requirements that have **not yet been validated.**

Metrics for the Requirements Model

Example: Software for System X has 24 individual functional requirements and 14 non-functional requirements. What is the specificity of the requirements? and completeness?

Specificity of the requirements : $\boldsymbol{Q_1 = n_{ui} / n_r}$

where

- n_{ui} = The no. of requirements for which all reviewers have similar understanding.
- $n_r = n_f + n_{nf} = 24 + 14 = 38$
- The closer the value of Q_1 to 1, the lower is the ambiguity of the specification.
- So, $1 = n_{ui} / 38 \rightarrow n_{ui} = 38$

Completeness of the requirements : $\boldsymbol{Q_2 = n_u / (n_i * n_s)}$

where

- n_u = The no. of unique functional requirements, $n_u = 24$
- n_i = The no. of inputs defined or implied by the specification
- n_s = The no. of states specified

Consider a Cab Booking project and its requirement details as follows, Assess **different specification metrics of requirement model** for the case study given above and show the final possible interpretation results.

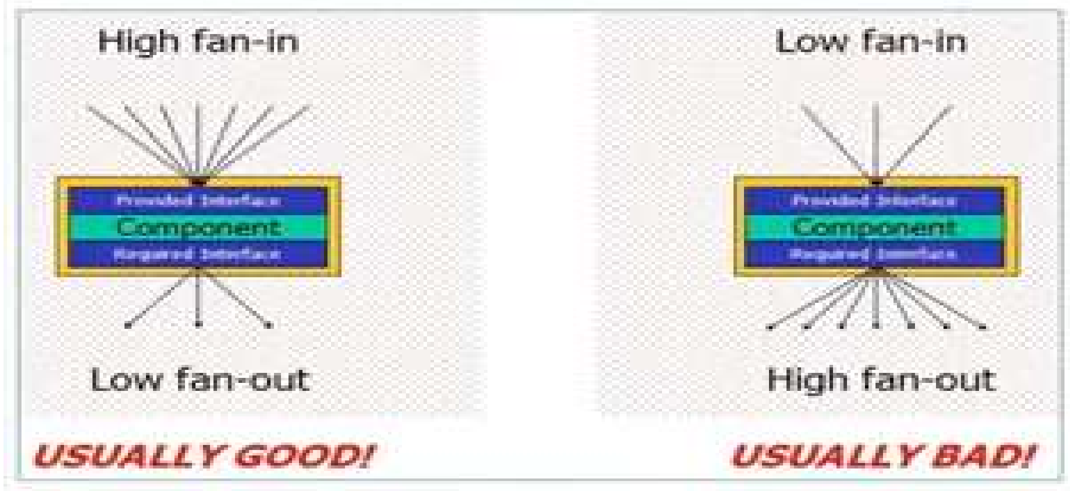
S.no	Specification	Values
1	Functional Requirements	20
2	Non-Functional Requirements	10
3	Requirements for which all reviewers had identical interpretations	15
4	Unique functional requirements	10
5	Inputs implied by the specification	5
6	States specified	2
7	Requirements that have been validated as correct	20
8	Requirements not yet been validated	10

Metrics for the Design Model

- The **success** of a software project depends **largely on the quality and effectiveness of the software design**.
- Hence, it is important to **develop software metrics for design** from which meaningful **indicators can be derived**.
  - With the help of indicators**, necessary steps** are taken **to design the software according to the user requirements.**
  - **Design metrics:**
    - Architectural design metrics,
    - Component-level design metrics,
    - User-interface design metrics, and
    - Metrics for object-oriented design

Metrics for the Design Model - Architectural design metrics

- **Focus** on the **characteristics of the program architecture, emphasis on architectural structure and effectiveness of components (or modules) within the architecture.**
- These metrics are **"black box"** in the sense that they **do not require any knowledge of the inner workings of a particular software component**.
- **Three software design complexity measures** are:
  - Structural complexity,
  - Data complexity, and
  - System complexity



Metrics for the Design Model - Architectural design metrics

Structural complexity:

- For **hierarchical architectures (e.g., call-and-return architectures),** structural complexity of a module **i** is $\mathbf{S(i) = f^2_{out}(i)}$

  Where fout(i) is the fan-out of module i.
- **Fan-out** : The no. of modules immediately subordinate to **module i**; (i.e., the number of modules that are directly invoked by module i)

Data complexity:

- Provides an **indication of the complexity in the internal interface** for a module i and $\mathbf{D(i) = v(i) / (f_{out}(i) + 1)}$
- Where v(i) = The no. of i/p and o/p variables passed to and from module i.

System complexity:

- The **sum of structural complexity and data complexity**: $\mathbf{C(i) = S(i) + D(i)}$

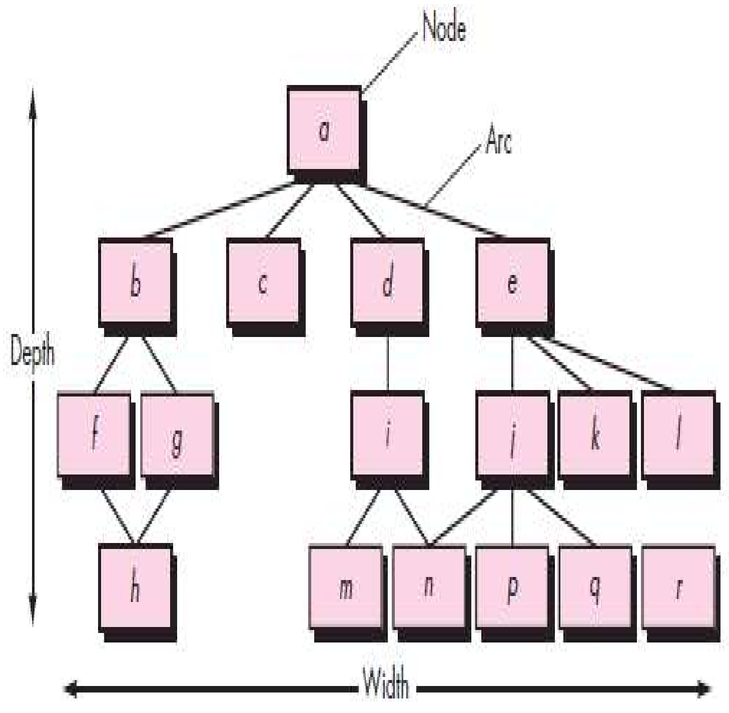
The complexity of a system **increases** with increase in structural complexity, data complexity, and system complexity, which in turn increases the integration and testing effort in the later stages.

Metrics for the Design Model - Architectural design metrics

- **Morphology (Shape) metrics:**
  - Allows **comparison of different program architectures using a set of straightforward dimensions.**
  - A function of the number of modules and the number of interfaces between modules.
    - Size
    - Depth
    - Width
    - The arc-to-node ratio

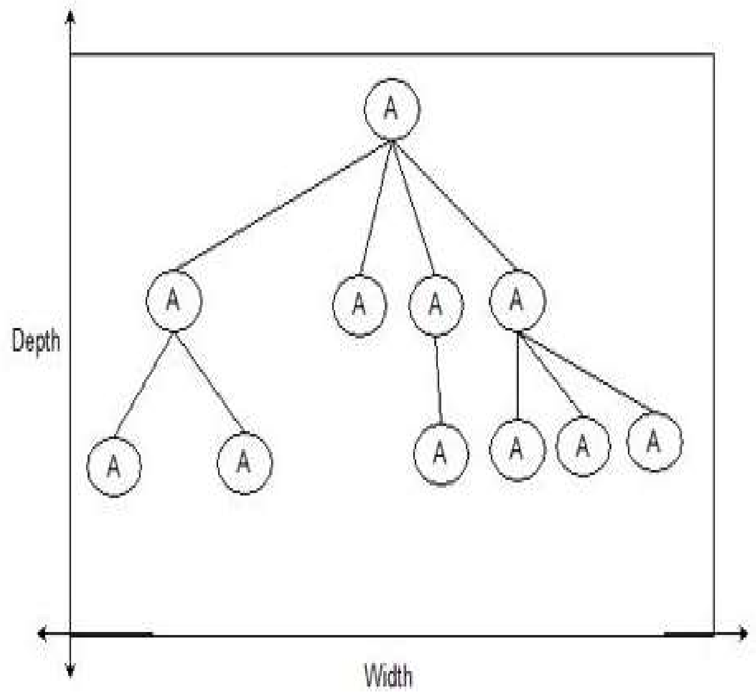
Metrics for the Design Model - Architectural design metrics

- **Size = n + a**,
  - where n is the number of nodes and a is the number of arcs
  - **Size = 17 + 18 = 35**
- **Depth = Longest path from the root (top) node to a leaf node**
  - Depth = 3
- **Width = Maximum number of nodes at any one level of the architecture**
  - Width = 6
- **The arc-to-node ratio, r = a/n**
  - Measures the **connectivity density** of the architecture and the coupling of the architecture,
  - → **r =** 18/17 = 1.06



Metrics for the Design Model - Architectural design metrics

- **Size = n + a**,
  - where n is the number of nodes and a is the number of arcs
  - **Size = 11 + 10 = 21**
- **Depth = Longest path from the root (top) node to a leaf node**
  - Depth = 2
- **Width = Maximum number of nodes at any one level of the architecture**
  - Width = 6
- **The arc-to-node ratio, r = a/n**
  - Measures the **connectivity density** of the architecture and the coupling of the architecture,
  - → **r** = 10/11 = 0.90



Metrics for the Design Model - Architectural design metrics

- **Quality** of **software design** plays an important role in determining the **overall quality of the software**.
- Many software quality indicators that are based on measurable design characteristics of a computer program have been proposed. One of them is **Design Structural Quality Index (DSQI),** which is **derived from the information obtained from data and architectural design.**
- DSQI ranges from **0 to 1.**
- To calculate DSQI,
  - *S*1 through *S*7 are determined for a computer program
  - Computer the intermediate values

Metrics for the Design Model - Architectural design metrics

To calculate DSQI, the following values must be determined:

- S1 = **Total no. of modules** defined in the program architecture
- S2 = No. of modules whose **correct function depends on the source of data input** or that produce data to be used elsewhere
- S3 = No. of modules whose **correct function depends on prior processing**
- S4 = No. of database items
- S5 = Total number of unique database items
- S6 = No. of database segments
- S7 = No. of modules with a single entry and exit

- Once values *S*1 through *S*7 are determined for a computer program,
- Next **computer the intermediate values**.

Metrics for the Design Model - Architectural design metrics

Calculation of Intermediate Values:

- **Program structure (D1):**
    - **D1 = 1** for discrete methods (e.g., data flow-oriented design or object –oriented design) are used for developing architectural design, else **D1 = 0**
- **Module independence (D2):** D2 = 1 - (S2 / S1)
- **Modules not dependent on prior processing (D3):** D3 = 1 - (S3 / S1)
- **Database size (D4):** D4 = 1 - (S5 / S4)
- **Database compartmentalization (D5):** D5 = 1 - (S6 / S4)
- **Module entrance / exit characteristic (D6):** D6 = 1 - (S7 / S1)

Calculation of DSQI:

$$\mathbf{DSQI = \sum W_i D_i}$$

where i = 1 to 6,

W_i = The relative weighting of the importance of each of the intermediate values, and $\sum W_i = 1$ (**If all D_i are weighted equally**, then $W_i = 0.167$).

Metrics for the Design Model - Architectural design metrics

A major information system has 1140 modules. There are 96 modules that perform control and coordination functions and 490 modules whose function depends on prior processing. The system processes approximately 220 data objects that each have an average of three attributes. There are 140 unique database items and 90 different database segments. Finally, 600 modules have single entry and exit points. Compute the DSQI for this system.

- S1 = 1140
- S2 = 96
- S3 = 490
- S4 = (220 * 3) = 660
- S5 = 140
- S6 = 90
- S7 = 600

- D1 = 1 [distinct method]
- D2 = 1 – (S2 / S1) = 1 – (96 / 1140) = 1 - 0.08 = 0.92
- D3 = 1 – (S3 / S1) = 1 – (490 / 1140) = 1 - 0.42 = 0.58
- D4 = 1 – (S5 / S4) = 1 – (140 / 660) = 1 - 0.21= 0.78
- D5 = 1 – (S6 / S4) = 1 – (90 / 660) = 1 - 0.14 = 0.86
- D6 = 1 – (S7 / S1) = 1 – (600 / 1140) = 1 - 0.52 = 0.48

- **Let us Assume W_i = 0.167**
- **DSQI = 0.167 * $\sum D_i$**
- **= 0.77**

Metrics for the Design Model - Architectural design metrics

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

Recap

- **Metrics for Requirements Model**
  - Function-Based Metrics (Function Point (FP) metric)
    - Based on FP → Productivity, documentation, cost per function
  - Metrics for Specification Quality
    - Metrics for Specificity, Completeness
- **Metrics for Design Model**
  - Architectural design metrics
    - Structural Complexity, Data Complexity and System Complexity
    - Morphology (Shape) metrics : Size, Depth, Width, Arc-to-Node Ratio
    - Design Structural Quality Index (DSQI)
  - Next
    - Metrics for object-oriented design
      - Class Oriented Metrics
    - Component-level design metrics

Metrics for the Design Model - Metrics for Object-Oriented Design

- The Object-Oriented (OO) paradigm is widely used in modern software engineering.
- OO design concepts such as such as classes, objects, polymorphism, encapsulation, inheritance, dynamic binding, information hiding, interface, constructor, destructor among others are used in the design methods.
- Most of the languages like C++, Java, .net use object oriented design concept.
- The main advantage of object oriented design is that **improving the software development and maintainability, faster and low cost development, and creates a high quality software.**
- The disadvantage of the object-oriented design is that larger program size and it is not suitable for all types of program.

Metrics for the Design Model - Metrics for Object-Oriented Design

Nine distinct and measurable characteristics of an OO design:

1. **Size:** Defined in terms of **four views**:
   - **I. Population** : Measured by taking a **static count of OO entities such as classes or operations.**
   - **II. Volume** : Measures are identical to population measures but are **collected dynamically - at a given instant of time.**
   - **III. Length :** Measure of a **chain of interconnected design elements** (e.g., the depth of an inheritance tree is a measure of length).
   - **IV. Functionality** metrics provide an **indirect indication** of the value delivered to the customer by an OO application.

Metrics for the Design Model - Metrics for Object-Oriented Design

2. **Complexity:** Determined by **assessing how classes are related to each other**.
3. **Coupling:** The **physical connection between OO design** elements
4. **Sufficiency:** The degree to which an **abstraction possesses the features required of it.**
5. **Completeness:** Similar to sufficiency, but has an **indirect implication about the degree to which the abstraction or design component can be reused**
6. **Cohesion:** Determined by **analyzing the degree** to which a set of properties that the class possesses is part of the problem domain or design domain
7. **Primitiveness:** Indicates the degree to which the **operation is atomic**
8. **Similarity:** Indicates similarity between **two or more classes in terms of their structure, function, behavior, or purpose**
9. **Volatility:** The probability of **occurrence of change** in the OO design

Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

- **CK - Chidamber and Kemerer**
- The class is the fundamental unit of an OO system.
- **Measures and metrics** for an individual class, the class hierarchy, and class collaborations will be helpful **to assess OO design quality**.
- **Six class-based design metrics** for OO systems:

1. **Weighted Methods per Class (WMC):**
   - Metric indicates the complexity of a class.
   - Measures the sum of complexity of the methods in a class
   - Assume $c_1, c_2, \ldots, c_n$ are complexities defined for a class C.
   - One way of calculating the complexity of a class is by using cyclomatic complexities of its methods.
   - $WMC = \sum c_i$ for $i = 1$ to n
   - WMC should be kept as low as possible, a higher value indicates that the class is more complex.

```
Class C{
    Method1{}
    Method2{}
    ;
    ;
    Methodn{}
}
```

Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

A class X has 12 operations. Cyclomatic complexity has been computed for all operations in the OO system, and the average value of module complexity is 4. For class X, the complexity for operations 1 to 12 is 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectively. Compute the weighted methods per class.

Answer:

WMC = $\sum c_i$ for i = 1 to n

Where c_i is individual complexity divided by the average complexity

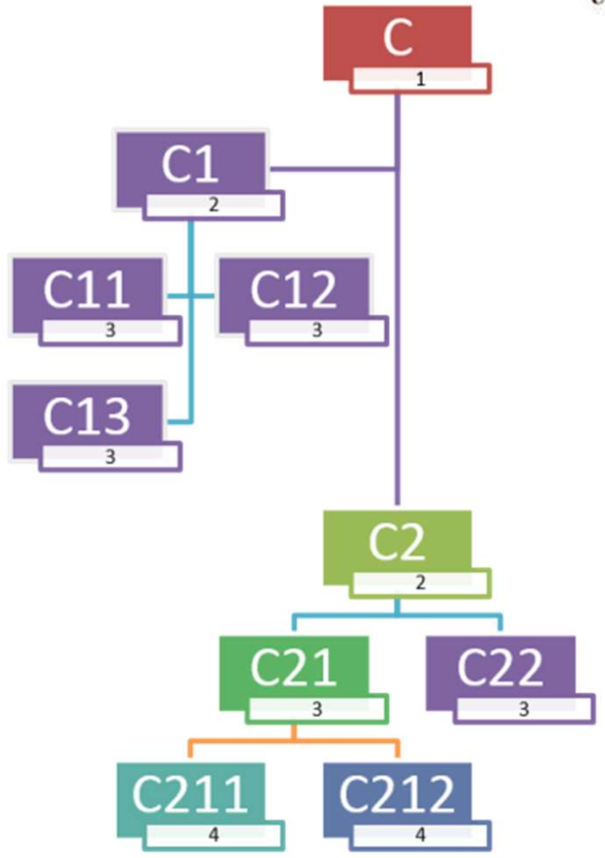
Average complexity is = 4

WMC = (5/4)+ (4/4)+ (3/4)+ (3/4)+ (6/4)+ (8/4)+ (2/4)+ (5/4)+ (5/4)+ (4/4)+ (4/4)

= 1.25 + 1 + 0.75 + 0.75 + 1.5 + 2 + 0.5 + 0.5 + 1.25 + 1.25 + 1 + 1 = 12.75

Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

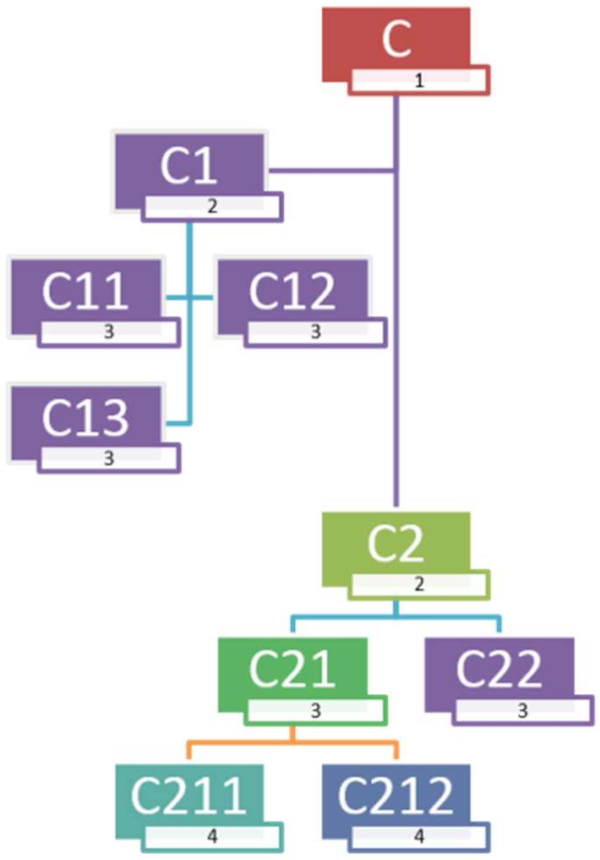
2. **Depth of the Inheritance Tree (DIT) :**
   - The **maximum length** from the **node to the root** of the tree
   - This metric indicates how far down a class is declared in the inheritance hierarchy.
   - DIT can be used as a measure of complexity of behavior of class, the complexity of design of a class and potential reuse.
   - A deep class hierarchy (DIT is large) also leads to greater design complexity.
   - Large DIT values imply that many methods may be reused



Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

3. Number of Children (NOC) :

- **Indicates** how many sub-classes are going to inherit the methods of the parent class.
- Class C2 has 2 children, subclasses C21, C22.
- As the number of children grows, reuse increases, but also, as NOC increases, the abstraction represented by the parent class can be diluted if some of the children are not appropriate members of the parent class.
- As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

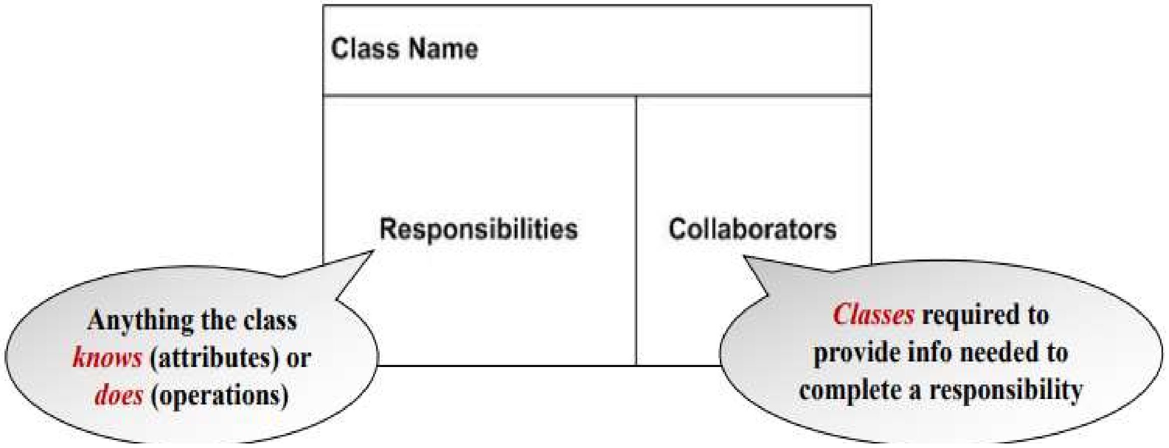


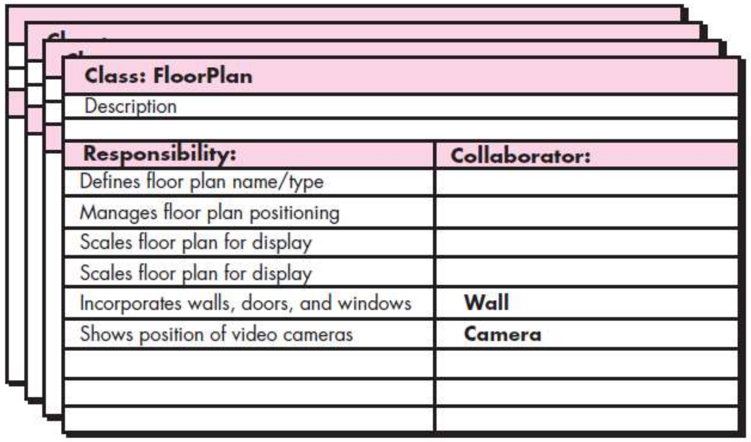
Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

4. **Coupling Between Object classes (CBO).**
   - The **Class-Responsibility-Collaborator (CRC) Model** may be used to determine the value for CBO.
   - A **CRC model** is a **collection of standard index cards** that represent classes, which are **divided into three sections**.
   - The **intent of CRC** is **to develop an organized representation of classes**

     Responsibilities → attributes and **operations** that are relevant for the class.

     Collaborators → classes that are **required** to provide a class with the information needed to complete a responsibility.


Class Name
Responsibilities
Collaborators
Anything the class *knows* (attributes) or *does* (operations)
Classes required to provide info needed to complete a responsibility



Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

Class-Responsibility-Collaborator (CRC) Mode

Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

4. **Coupling Between Object classes (CBO).**
    - The motivation behind this metric is that **an object is coupled to another object if two object acts upon each other**.
    - If a class **uses** the methods of other classes, then they both are coupled.
    - An increase in CBO indicates an increase in responsibilities of a class.
    - Hence, the CBO value for classes should be kept as low as possible.

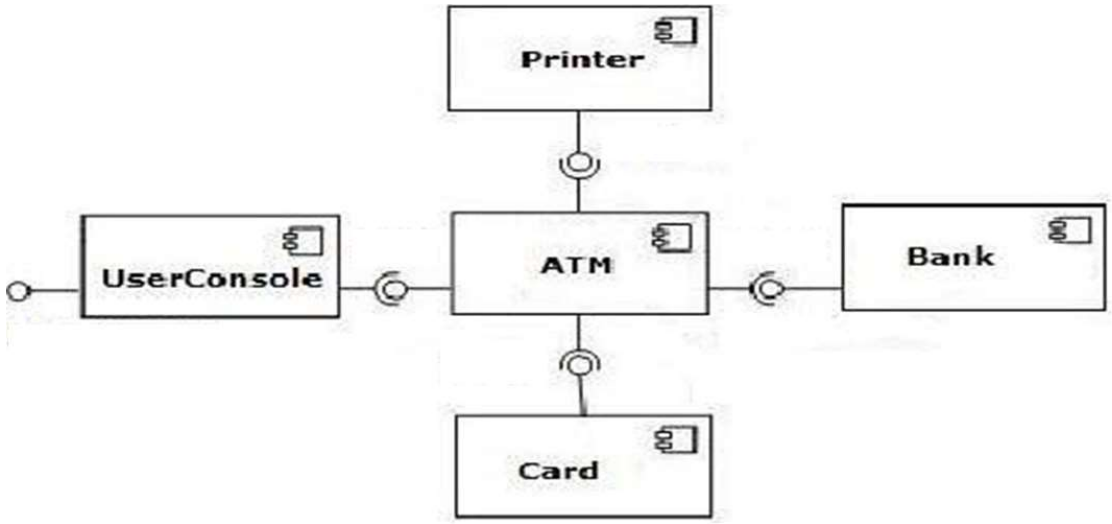
5. **Response For a Class (RFC):**
    - A **set of methods** that can potentially be **executed** in **response to a message received** by **an object** of that class.
    - As RFC increases, the effort required for testing also increases because the test sequence grows and the overall design complexity of the class increases.

Metrics for the Design Model –
Class-Oriented Metrics—The CK Metrics Suite

6. **Lack of COhesion in Methods (LCOM).**
   - **Cohesion :** An indication of the **relative functional strength** of a module
   - Each method within a class C accesses one or more attributes (also called instance variables).
   - LCOM is the number of methods that access one or more of the same attributes.
     - If no methods access the same attributes, then LCOM = 0.
     - If a class is having six methods, four of the methods have one or more attributes in common (i.e. they access common attributes). Therefore, LCOM = 4.

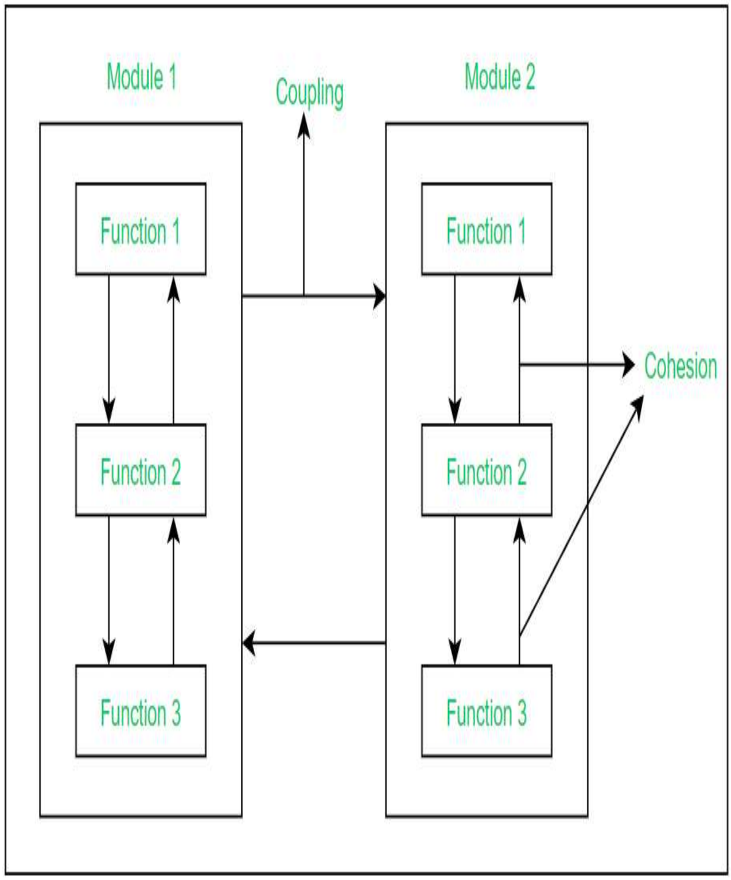
Metrics for the Design Model–Component-Level Design Metrics

- **Focus** on **internal characteristics** of a software component and include measures of the “three Cs”- module **Cohesion**, **Coupling**, and **Complexity**.
- These measures can help **to judge the quality of a component-level design**.
- These metrics are **glass box - require knowledge of the inner working of the module** under consideration.
- Component-level design metrics may be applied once a procedural design has been developed.



Metrics for the Design Model–Component-Level Design Metrics

- **Cohesion**
  - An indication of the **relative functional strength** of a module
  - Cohesion is an **extension** of the information hiding concept.
  - A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.
- **Coupling**
  - An **indication of interconnection between modules** in a structure of software (measures the relative interdependence among modules).



A good software design requires **high cohesion and low coupling**.

Metrics for the Design Model–Component-Level Design Metrics

- **Cohesion metrics:**
  - **Define** a collection of metrics that **provide an indication of the cohesiveness of a module.**
  - The metrics are defined in terms of **five concepts and measures:**
    - **Data tokens:** The variables or constants defined for a module
    - **Slice:** The **collection of all the statements** with in a program **that can affect the value of some specific variable** of interest.
    - **Data Slice:** The **collection of all the data tokens in the slice** that will affect the value of a specific variable of interest. Both program slices (which focus on statement and conditions) and data slices can be defined.
    - **Glue tokens**: The set of data tokens lies on one or more data slice.
    - **Superglue tokens:** The set of tokens on all slices
    - **Stickiness**: The relative stickiness of a glue token is directly proportional to the number of data slices that it binds.

Measure Program Cohesion through **2 metrics:-**

Weak functional cohesion = (# of glue tokens) / (total # of data tokens)

Strong functional cohesion = (#of super glue tokens) / (total # of data tokens)

Metrics for the Design Model–Component-Level Design Metrics

```
MinMax ( z, n)
Integer  end, min, max, i ;
end = n ;
max = z[0] ;
min = z[0] ;
For ( i = 0, i = < end , i++ ) {
      if  z[ i ] > max then max = z[ i ];
      if  z[ i ] < min then min = z[ i ];
                                            }
  return max, min;
```

Data Tokens:
z1
n1
end1
min1
max1
i1
end2
n2
max2
z2
01
min2
z3
02
i2
03
i3
end3
i4
z4
i5
max3
max4
z5
i6
z6
i7
min3
min4
z7
i8
max5
min5 (33)

Slice max:
z1
n1
end1
max1
i1
end2
n2
max2
z2
01
i2
03
i3
end3
i4
z4
i5
max3
max4
z5
i6
max5
(22)

Slice min:
z1
n1
end1
min1
i1
end2
n2
min2
z3
02
i2
03
i3
end3
i4
z6
i7
min3
min4
z7
i8
min5
(22)

Glue Tokens:
z1
n1
end1
i1
end2
n2
i2
03
i3
end3
i4 (11)

Super Glue:
z1
n1
end1
i1
end2
n2
i2
03
i3
end3
i4 (11)

The Cohesion **metrics (MinMax):-**
Weak functional cohesion = 11 / 33 = 1/3
Strong functional cohesion = 11 / 33 = 1/3

Metrics for the Design Model–Component-Level Design Metrics

- **Coupling metrics: A function of input and output parameters, global variables and modules called.**
  - Module coupling **provides an indication of the connectedness of a module to other modules, global data, and the outside environment.**
  - Metric for module coupling includes data and control flow coupling, global coupling, and environmental coupling.

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)

Metrics for the Design Model–Component-Level Design Metrics

Module Coupling Indicator: m_c = k / M

where k is a proportionality constant (k = 1) and

$M = (d_i + a * c_i + d_0 + b * c_0 + g_d + c * g_c + w + r)$

Values for k, a, b, and c must be derived empirically. k = 1, a=b=c=2

- Higher the value of m_c, the lower the overall module coupling.
- **Example:** If a module has single input and output data parameters, accesses no global data, and is called by a single module,
    - d_i , d_0 = 1 $c_{i,}$ c_0 = 0, g_d , g_c = 0, w = 1, and r = 0
    - m_c = 1/(1 + 0 + 1+ 0 + 0 + + 0 + 1 + 0) = 1/3 = 0.33
    - Module exhibits low coupling →mc = 0.33 implies low coupling
- **Example:** If a module has five input and five output data parameters, an equal number of control parameters, accesses ten items of global data, has a fan-in of 3 and a fan-out of 4,
    - d_i =5, d_0 = 5 $c_{i,}$ c_0 = 5, g_d = 10, g_c = 0, w = 4, and r = 3
    - mc = 1/(5 + 2*5 + 5 + 2*5 + 10 + 0 + 4 + 3) = 0.02 (High Coupling)
- As the value for $\mathbf{m_c}$ increases, the overall module coupling decreases.
- Coupling Metric C = 1 - $\mathbf{m_c}$ where the **degree of coupling increases** as the value of M increases.

Metrics for the Design Model–Component-Level Design Metrics

Complexity metrics:

- A variety of software metrics can be computed to determine the complexity of program control flow.
- The most widely used complexity metric for computer software is **cyclomatic complexity**, developed by Thomas McCabe.

Metrics for the Design Model–Component-Level Design Metrics

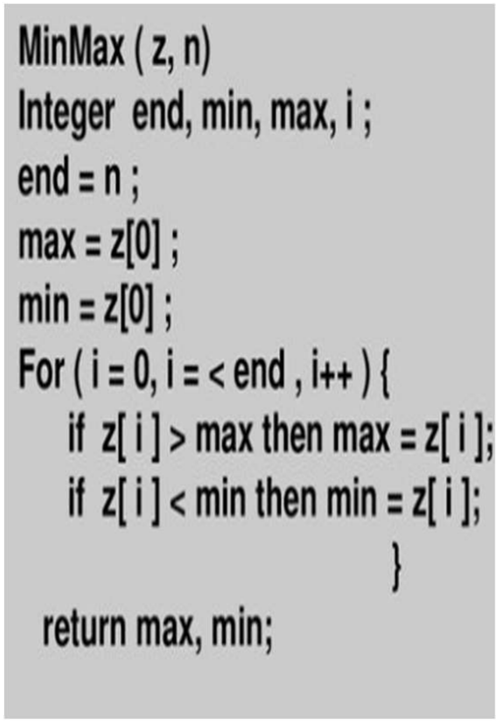
- **Cohesion metrics:**
  - **Define** a collection of metrics that **provide an indication of the cohesiveness of a module.**
  - The metrics are defined in terms of **five concepts and measures:**
    - **Data tokens:** The variables or constants defined for a module
    - **Slice:** The **collection of all the statements** with in a program **that can affect the value of some specific variable** of interest.
    - **Data Slice:** The **collection of all the data tokens in the slice** that will affect the value of a **specific variable of interest**. Both program slices (which focus on statement and conditions) and data slices can be defined.
    - **Glue tokens**: The set of data tokens lies on **one or more data slice**.
    - **Superglue tokens:** The set of tokens on **all slices.**
    - **Stickiness**: The relative stickiness of a glue token is directly proportional to the number of data slices that it binds.

Measure Program Cohesion through 2 metrics:-

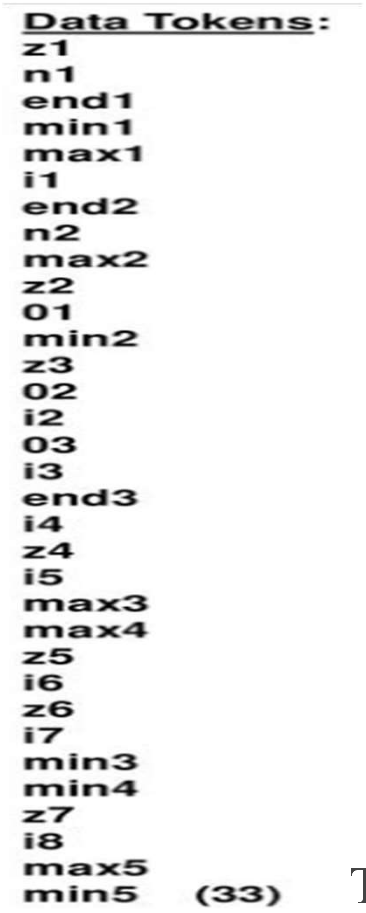
Weak functional cohesion = (# of glue tokens) / (total # of data tokens)

Strong functional cohesion = (#of super glue tokens) / (total # of data tokens)

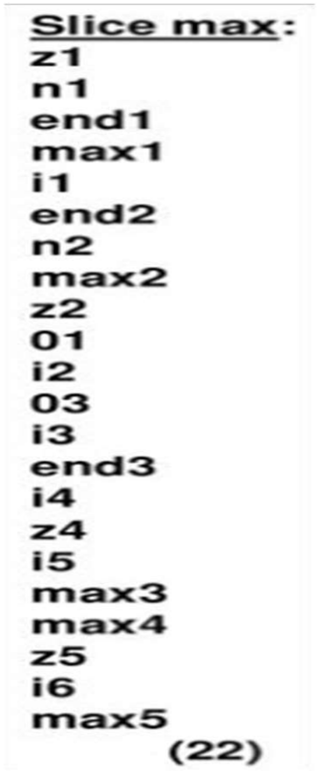
Metrics for the Design Model–Component-Level Design Metrics



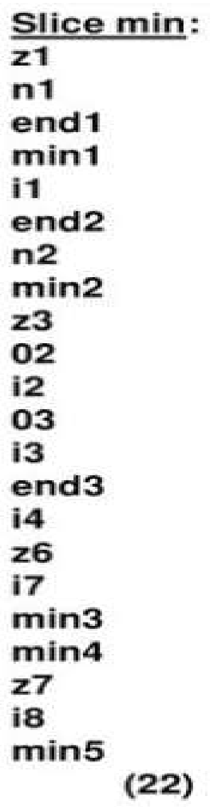
```
MinMax ( z, n)
Integer  end, min, max, i ;
end = n ;
max = z[0] ;
min = z[0] ;
For ( i = 0, i = < end , i++ ) {
    if  z[ i ] > max then max = z[ i ];
    if  z[ i ] < min then min = z[ i ];
                              }
 return max, min;
```



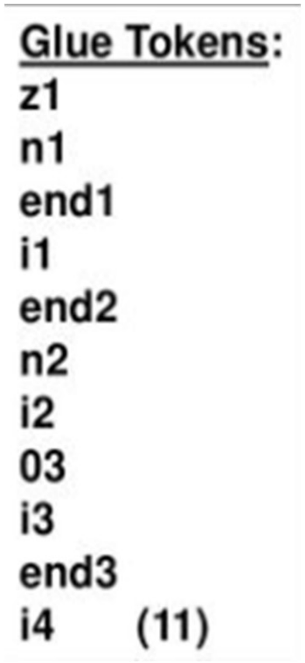
Data Tokens:
z1
n1
end1
min1
max1
i1
end2
n2
max2
z2
01
min2
z3
02
i2
03
i3
end3
i4
z4
i5
max3
max4
z5
i6
z6
i7
min3
min4
z7
i8
max5
min5 (33)



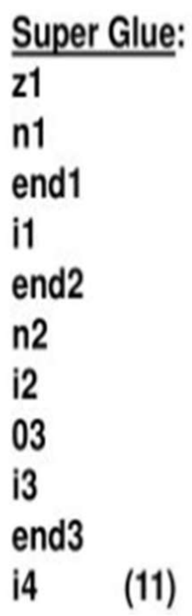
Slice max:
z1
n1
end1
max1
i1
end2
n2
max2
z2
01
i2
03
i3
end3
i4
z4
i5
max3
max4
z5
i6
max5
(22)



Slice min:
z1
n1
end1
min1
i1
end2
n2
min2
z3
02
i2
03
i3
end3
i4
z6
i7
min3
min4
z7
i8
min5
(22)



Glue Tokens:
z1
n1
end1
i1
end2
n2
i2
03
i3
end3
i4 (11)



Super Glue:
z1
n1
end1
i1
end2
n2
i2
03
i3
end3
i4 (11)

The Cohesion **metrics (MinMax):-**
Weak functional cohesion = 11 / 33 = 1/3
Strong functional cohesion = 11 / 33 = 1/3

Metrics for the Design Model–Component-Level Design Metrics

```
int n;
int sum;
int prod;
int test;
sum = 0;
prod = 1;
test = prod;
for(int i=1; i<=n; i++)
{
sum = sum + i;
prod = prod*i;
test = test+sum * prod;
}
```

sum	prod	test	Data tokens	
n1	n1	n1	1	int n;
sum1			1	int sum;
	prod1		1	int prod;
		test1	1	int test;
sum2, 01			2	sum = 0;
	prod2 11		2	prod = 1;
	test2, prod3	test2 prod3	2	test = prod;
i1, 12, i2, n2,i3	i1, 12, i2, n2,i3	i1, 12, i2, n2,i3	5	for(int i=1; i<=n; i++){
sum3 sum4, i4			3	sum = sum + i;
	prod3 prod4 i5		3	prod = prod * i;
sum5	prod5	test3 test4 sum5,prod5	4	test = test + sum * prod; }

Superglue tokens: n1, i1, 12, i2, n2,i3 = 6
Glue tokens: n1, Test2, Prod3, i1, 12, i2, n2, i3, sum5, prod5

Tokens = 25
Weak functional cohesion = 10 / 25
Strong functional cohesion = 6 / 25

Recap

- **Metrics for Requirements Model**
  - Function-Based Metrics (Function Point (FP) metric)
    - Based on FP → Productivity, documentation, cost per function
  - Metrics for Specification Quality
    - Metrics for Specificity, Completeness
- **Metrics for Design Model**
  - Architectural design metrics
    - Structural Complexity, Data Complexity and System Complexity
    - Morphology (Shape) metrics : Size, Depth, Width, Arc-to-Node Ratio
    - Design Structural Quality Index (DSQI)
    - Metrics for object-oriented design
      - Class Oriented Metrics
    - Component-level design metrics
- **Metrics for Software Quality**

Metrics for Software Quality

- The **prime goal** of software engineering is to produce a **high-quality system, application, or product** within a time frame that satisfies a market need.
- To achieve this goal,
  - **Apply effective methods** coupled with modern tools within the context of a mature software process.
  - In addition, a good software engineers or managers must measure if high quality is to be realized.
- The **quality** of an **application** is only as good as
  - **Requirements → Describe the problem**,
  - **Design → Models the solution**,
  - **Code → Leads to an executable program**, and
  - **Tests → Exercise the software to uncover errors**.

Measuring Quality Indicators

- Correctness,
- Maintainability,
- Integrity,
- Usability

Metrics for Software Quality

- **Correctness:**
  - Correctness is the **degree to which the software performs the required functions accurately or correctly.**
  - **Defects per KLOC** (Thousands (Kilo) of Lines of Code) is one of the common measure.
  - KLOC is a way of measuring the size of a computer program by **counting the number of lines of source code a program has.**
- **Maintainability:**
  - The ease with which a **program can be correct if an error is occurred**.
  - Since there is **no direct way** of measuring maintainability, an **indirect measure** like **MTTC (Mean Time To Change)** is used.
  - It measures when a error is found, how much time it takes to analyze the change, design the modification, implement it and test it.

Metrics for Software Quality

Integrity:

- Measures a system's **ability to with stand attacks to its security**.
- Increasingly important in the age of cyber terrorists and hackers.
- Attacks can be made on all three components of software: Programs, Data, and Documentation.
- Parameters required to measure integrity:
  - **Threat -** Probability that an attack of certain type **will happen** over a period of time.
  - **Security -** Probability that an attack of certain type **will be removed** over a period of time.

$$\textbf{Integrity} = \sum [1 - (\text{threat X } (1 - \text{security}))]$$

Example:

- If threat (the probability that an attack will occur) is 0.25 and security (the likelihood of repelling an attack) is 0.95, the integrity of the system is 0.99 (very high).
- On the other hand, the threat probability is 0.50 and the likelihood of repelling an attack is only 0.25, the integrity of the system is 0.63 (unacceptably low).

Metrics for Software Quality

Example:

- A WebApp and its support environment have not been fully fortified against attack. Web engineers estimate that the likelihood of repelling an attack is only 30 percent. The system does not contain sensitive or controversial information, so the threat probability is 25 percent. What is the integrity of the WebApp?

Answer:

- Repelling attack: 30%
- Threat Probability: 25 %
- **Integrity = ∑ [1 – (threat X (1 - security))]**

  = ∑ [1 – (0.25 X (1 – 0.3))]

  = [1 – (0.25 X 0.7)]

  = 1-0.18

  = 0.83

Metrics for Software Quality

Usability:

- If a program is not easy to use, it is often doomed to failure, even if the functions that it performs are valuable.
- Usability is **an attempt to quantify ease of use** and can be measured in terms of the following characteristics:
  - Physical / Intellectual skill required to learn the system
  - Time required to become moderately efficient in the system.
  - The net increase in productivity by use of the new system.
  - Subjective assessment(usually in the form of a questionnaire on the new system)

Metrics for Software Quality

Defect Removal Efficiency (DRE):

- Used to **assess the ability of a team to determine errors before the errors are passes to the upcoming engineering activity**
- DRE is a **measure of the efficacy** of Software Quality Assurance activities.

$$\mathbf{DRE = E / (E + D)}$$

Where

- E = The number of errors found before delivery of the software to the end user
- D = The number of defects found after delivery

- The ideal value for DRE is 1. i.e., no defects are found in the software.
- If score low on DRE → need to re-look at existing process.
- It encourages the team to find as many defects before they are passed to the next activity stage.

Metrics for Software Quality

Defect Removal Efficiency (DRE):

- DRE can also be used **within the project to assess a team's ability to find errors** before they are passed to the next framework activity or software engineering action.
- Example:
  - Requirements analysis produces a requirements model that can be reviewed to find and correct errors. Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found).
  - Redefine DRE as

$$\mathbf{DRE_i = E_i / (E_i + E_{i+1})}$$

Where

- E_i = The no. of errors found during software engineering action i
- E_{i+1} = The no. of errors found during software engineering action i + 1 that are traceable to errors that were not discovered in software engineering action i.

Metrics for Software Quality

Example:

At the conclusion of a project, it has been determined that 30 errors were found during the modeling phase and 12 errors were found during the construction phase that were traceable to errors that were not discovered in the modeling phase. What is the DRE for these two phases?

Answer:

- The DRE or defect removal efficiency is used to assess the ability of a team to determine errors before the errors are passes o the upcoming engineering activity. DRE is thus defined as:

  $\mathbf{DRE_i = E_i / (E_i + E_{i+1})}$

  Given $E_i = 30$, $E_{i+1} = 12$

  $DRE_i = 30 / (30 + 12) \rightarrow 30 / 42 \rightarrow 0.7143$

Metrics for Software Quality

Example: A software team delivers a software increment to end users. The users uncover eight defects during the first month of use. Prior to delivery, the software team found 242 errors during formal technical reviews and all testing tasks. What is the overall DRE for the project after one month's usage?

Defects found during the first month of use = 8
The errors found during formal technical reviews = 242

Errors uncovered during work to make change = 242
Defects uncovered after change is released to the customer base = 8
$DRE_i = E_i / (E_i + E_{i+1})$
$= 242 / (242 + 8) \rightarrow 242 / 250 = 0.968$

Metrics for Software Quality

Example: Suppose the total number of defects logged is 500, out of which 20 were found after delivery, and 200 were found during the system testing.

- The defect removal efficiency of system testing is 200/220 (just about 90%), as the total number of defects present in the system when testing started was 220.

- The defect removal efficiency of the overall quality process is 480/500, which is 96%.

Metrics for Software Quality

Example: Suppose the total number of defects logged is 500, out of which 20 were found after delivery, and 200 were found during the system testing.

- The defect removal efficiency of system testing is 200/220 (just about 90%), as the total number of defects present in the system when testing started was 220.
- The defect removal efficiency of the overall quality process is 480/500, which is 96%.

Metrics for Maintenance - Software Maturity Index (SMI):

Provides an indication of the **stability of a software product** (based on changes that occur for each release of the product).

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

Where

M_T = Number of modules in the **current release**

F_c= Number of modules in the **current release that have been <u>c</u>hanged**

F_a = Number of modules in the **current release that have been <u>a</u>dded**

F_d= Number of modules from preceding release that were **<u>d</u>eleted** in the **current release**

- As SMI approaches 1.0, the product begins to stabilize.
- SMI may be used for planning software maintenance activities.

Metrics for Maintenance

A legacy software system has 940 modules. The latest release require that 90 of these modules be changed. In addition, 40 new modules were added and 12 old modules were removed. Compute the software maturity index for the system.

Answer:

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

Where

M_T = Number of modules in the **current release = 940**

F_c= Number of modules in the **current release that have been changed = 90**

F_a = Number of modules in the **current release that have been added = 40**

F_d= Number of modules from preceding release that were **deleted** in the **current release = 12**

SMI = (940 - (40 + 90 + 12)) /940

= 0.8489

= 0.849