Measure Software Quality

How can we know ...

- How good is a program?
- How reliable will a software system be once it is installed?
- How much more testing should I do?
- How many defects can I expect to find?
- How much will the testing cost?
- How easy will it be to maintain a system?
- How long will it take to conduct the integration for the system?

Measurement & Metrics

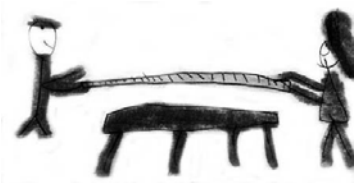
- Collecting metrics is too hard …
- It's too time-consuming …
- It won't prove anything …
- It's too political …

Anything that you need to quantify can be measured in some way that is superior to not measuring it at all ..

Tom Gilb

Why do we Measure?

- To Characterize
- To Evaluate
- To Predict
- To Improve



Software Metrics

- Software Metrics refers to a broad range of measurements in computer software for a variety of purposes:
  - Applied to the software processes with the intent of improving it on a continual basis.
  - Used throughout a software project to assist in Estimation, Quality Control, Productivity Assessment, and Project Control.
  - Used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds.

Main Purposes of Measurement

- **A**ssessment and certification of the product
- **M**aintenance cost estimation
- **I**mprovement of the development process
- **C**ontrol of the project
- **E**stimation of future projects

Main Purposes of Measurement – Assessment and certification of the product

- Before delivery to the customer, we must know if the product is fit for its purpose.

Main Purposes of Measurement – Maintenance Cost Estimation

- After delivery, software must be maintained. Faults found in operation need to be corrected, and the software may need to be enhanced or adapted to new environments.

Main Purposes of Measurement – Improvement of the development process

- Having measured the quality of the software produced using particular development methods, and compared this to the levels of quality achieved using different methods, and adopt the better ones for future projects.

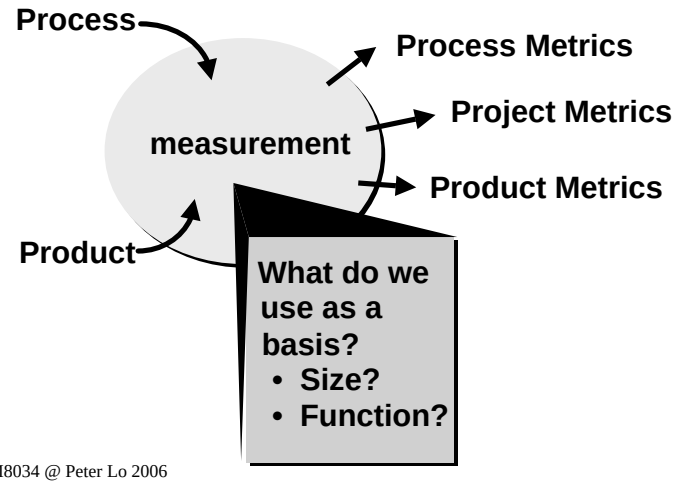
Main Purposes of Measurement – Control of the Project

- In order to know if the current project is staying within its constraints (keeping within budget, on target to meet the delivery deadline, etc), we must measure such things as the amount of progress and expenditure so far.

Main Purposes of Measurement – Estimation of Future Projects

- We base the estimate of cost and schedule of a new project on measurements made of cost, size, and development time of previous projects.

A Good Manager Measures



Process Metrics

- Majority focus on quality achieved as a consequence of a repeatable or managed process
- Statistical SQA data
  - Error categorization & analysis
- Defect removal efficiency
  - Propagation from phase to phase
- Reuse data

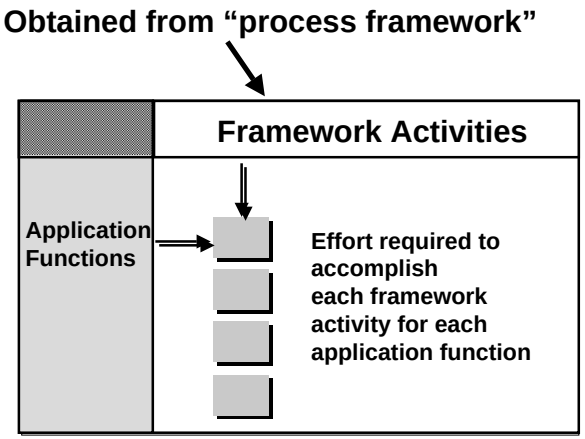
Project Metrics

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

Product Metrics

- Focus on the quality of deliverables
- Measures of analysis model
- Complexity of the design
  - Internal algorithmic complexity
  - Architectural complexity
  - Data flow complexity
- Code measures (e.g., Halstead)
- Measures of process effectiveness
  - E.g. defect removal efficiency

Creating a Task Matrix



Metrics Guidelines

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who have worked to collect measures and metrics.
- Don’t use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered “negative.” These data are merely an indicator for process improvement.
- Don’t obsess on a single metric to the exclusion of other important metrics.

Measures, Metrics, and Indicators

- A **Measure** is established when a single data point is collected.
- A **Software Metric** relates individual measures in a meaningful way. (e.g., the average number of errors found per review or the average number of errors found per person-hour spend on reviews).
- An **Indicator** is a metric or combination of metrics that provide insight into the software project, process, or product.

Measures, Metrics, and Indicators

- A software engineer collects measures and develops metrics so that indicators will be obtained.
- An indicator provides insight that enables the software engineers to adjust the process, the project, or the product to make things better.

Process Indicators & Process Metrics

- **Process Indicators** enable a software engineering organisation to gain insight into the efficacy of an existing process. They enable managers and practitioners to assess what works and what does not.
- **Process Metrics** are collected across all projects and over long period of time. Their intent is to provide indicators that lead to long-term software process improvement.

Process Indicators

- Project indicators enable software engineer to
  - Assess the status of an ongoing project
  - Track potential risks
  - Uncover problem areas before they "go critical"
  - Adjust work flow or tasks
  - evaluate the project team's ability to control quality of software engineering work products.

Fundamentals of Measurement Theory

- Measurement is the act of determining a measure.
- Measurement occurs as the result of the collection of one or more data points.
  - An **Entity** is any object (or process, or activity) in the real world, which we need to describe.
  - An **Attribute** is any property or characteristic of an entity that we can observe.
- Measurement describes entities by assigning numbers to their attributes.
- The result is a mapping of entities into some number system.
- Numbers can be real or integers, or a set of arbitrary symbols among which no order is defined

Processes, Resources and Products

- When managing a software development project, we are interested in the following entity:
  - **Product**: output of a manufacturing process
    - E.g. Source code, Requirements specification document
  - **Process**: activity which consumes resource and generates product
    - E.g. Writing code, Capturing requirements
  - **Resource**: consumed by a manufacturing process
    - E.g. Effort, Consumables (paper, electrical power, etc)

Processes, Resources and Products

- A manufacturing process is an activity in which resources are invested in order to generate a product.
- In order to make meaningful measurements, we must
  - Select entities
  - Decide which attributes to measure
  - Define each attribute clearly and quantitatively
  - Choose appropriate scales

Types of Attribute

- **Internal**: relate to entity in isolation, e.g. size of source code, age of personnel.
- **External**: depend on relationship of entity to environment, e.g. reliability of object code, stability of requirement.
- We access software by how well its function suits our needs.
- We judge software on its external attributes.
- We measure how well it satisfies us by defining certain attributes in a way which allows us to make measurements by observing it in use. E.g. reliability.

Software Measurement

- **Direct Measures**
  - Lines of Code (LOC) produced
  - Execution speed
  - Memory size
  - Defects reported over some period of time
- **Indirect Measures**
  - Functionality
  - Quality
  - Complexity
  - Efficiency
  - Reliability and maintainability

Software Measurement

- **Static Measures**
  - Derived from examination of the software itself, e.g., source code.
- **Dynamic Measure**
  - Derived from observations of the execution of the software

Software Measurement

- **Effort Measures**
  - Most common units of measurement of effort are labour-month, staff week, staff-month, person-year.
- **Quality Measures**
  - Two fundamental ideas related to quality are freedom from defect and suitability for use.
  - This suggests quality measures should be counts of defects and problem reports.

Software Measurement

- **Performance Measures**
  - Two common performance measures:
    - Response Time – how long it takes to accomplish a task
    - Throughput – tasks can be completed in a unit of time
- **Reliability Measures**
  - Software reliability cannot be measured directly. It is generally computed from other measures of the behaviour of the software.

Analysis Metrics

- **Function-based Metrics**
  - Use the function point as a normalizing factor or as a measure of the "size" of the specification
- **Bang Metric**
  - Used to develop an indication of software "size" by measuring characteristics of the data, functional and behavioral models
- **Specification Metrics**
  - Used as an indication of quality by measuring number of requirements by type

Architectural Design Metrics

- **Architectural Design Metrics**
  - Structural Complexity = g(fan-out)
  - Data Complexity = f(input & output variables, fan-out)
  - System Complexity = h(structural & data complexity)
- **HK Metric**
  - Architectural complexity as a function of fan-in and fan-out
- **Morphology Metrics**
  - A function of the number of modules and the number of interfaces between modules

Component-Level Design Metrics

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

Interface Design Metrics

- **Layout Appropriateness**
  - A function of layout entities, the geographic position and the "cost" of making transitions among entities

Code Metrics

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

Normalization for Metrics

- Normalized data are used to evaluate the process and the product (but never individual people)
  - **Size-oriented Normalization** – the Line of Code (LOC) approach
  - **Function-oriented Normalization** – the Function Point (FP) approach

Typical Size-Oriented Metrics

- LOC measures are programming language dependent.
  - LOC (or KLOC)
  - Errors per KLOC (thousand lines of code)
  - Defects per KLOC
  - $ per LOC
  - Pages of documentation per KLOC
  - Errors / person - month
  - LOC per person -month
  - $/page of documentation

Sized-Oriented Metrics

- Consider the "size" of the software
- A common factor or normalization value can be used as Lines of Code (LOC).
- Size oriented metrics can now include: Errors per KLOC (thousand lines of code), Defects per KLOC, Cost per LOC, Pages of documentation per KLOC.

Project	LOC	Effort	$(000)	pp. doc.	Errors	Defects	People
Alpha	12,100	24	168	365	134	29	3
Beta	27,200	62	440	1224	321	86	5
Gamma	20,200	43	314	1050	256	64	6
(etc.)							

Typical Function-Oriented Metrics

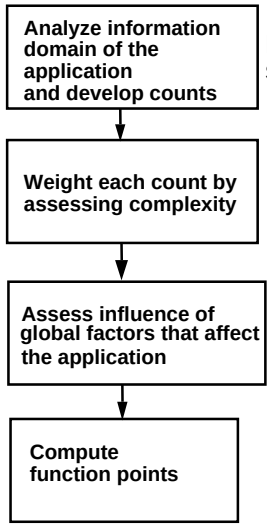
- Function points are derived using an empirical relationship based on (direct) measures of software's information domain and assessments of software complexity.
  - Errors per FP (thousand lines of code)
  - Defects per FP
  - $ per FP
  - Pages of documentation per FP
  - FP per person-month

Function-Oriented Metrics

- Use a measure of the functionality delivered by the application as a normalization value.
- Based on the calculated value of function points, can then be used like LOC to define software measures:
  - Errors per FP
  - Defects per FP
  - $ per FP
  - Pages of documentation per FP

$$FP = counttotal \times [0.65 + (0.01 \times \sum F_i)]$$

Computing Function Points



Establish *count* for input domain and system interfaces

Assign level of complexity or *weight* to each count

Grade significance of external factors, F such as reuse, concurrency, OS, ...

function points = $\sum$ (count x weight) x C

where:

complexity multiplier: C = (0.65 + 0.01 x N)

degree of influence: N = $\sum_i$ F

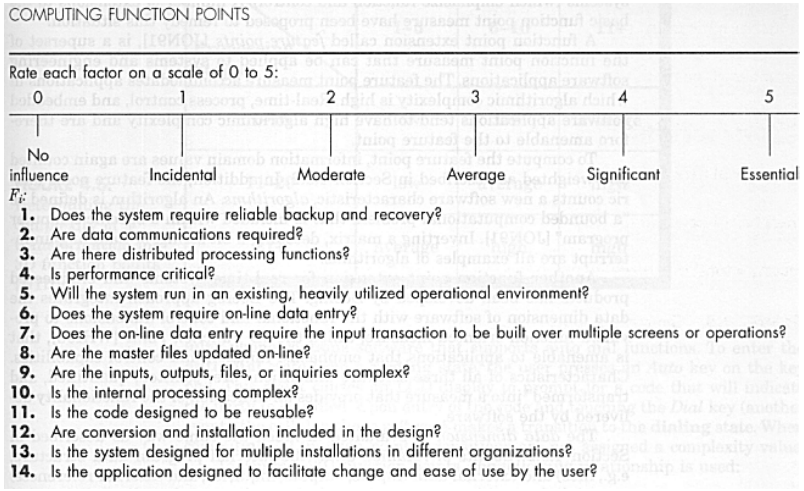
Function-Oriented Metrics

- Counttotal is the sum of all entries

measurement parameter	count		Weighting Factor simple	average	complex	
number of user inputs		×	3	4	6	=
number of user outputs		×	4	5	7	=
number of user inquiries		×	3	4	6	=
number of files		×	7	10	15	=
number of external interfaces		×	5	7	10	=
count = total →						

Function-Oriented Metrics

- F(i) (where i = 1 to 14) are "complexity adjustment values".


COMPUTING FUNCTION POINTS

Rate each factor on a scale of 0 to 5:

0 (No influence) 1 (Incidental) 2 (Moderate) 3 (Average) 4 (Significant) 5 (Essential)

F_i:

1. Does the system require reliable backup and recovery?
2. Are data communications required?
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 on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
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?

Example

- For system Z:
  - Does the system require reliable backup and recovery? ➔ Average (3)
  - Are data communications required? ➔ Essential (5)
  - Is performance critical? ➔ Incidental (1)
- The summation for F(i) would be 3 + 5 + 1 = 9 for the first three values then.
- FP = 40 x [0.65 + (0.01 x 52)] = 46.8

Function Points

- A set of function-oriented metrics can be developed for each project from the following data:
  - Errors per FP
  - Defects per FP
  - $ per FP
  - Pages of documentation per FP
  - FP per person - month

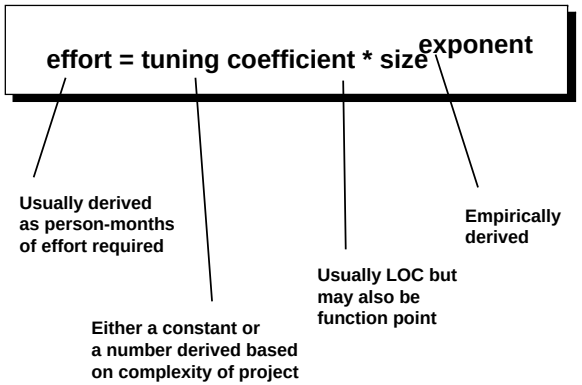
More Function Points Examples

- Number of user inputs
  - Input that provides distinct application-oriented data.
- Number of user outputs
  - Reports, screens, error messages etc. NOT individual data items.
- Number of user inquires
  - On-line input that generate immediate software response.
- Number of files
  - Logical master file.
- Number of external interfaces
  - Interface uses to transmit information to another system.

Why Opt for FP Measures?

- Independent of programming language
- Uses readily countable characteristics of the "information domain" of the problem
- Does not "penalize" inventive implementations that require fewer LOC than others
- Makes it easier to accommodate reuse and the trend toward object-oriented approaches

Empirical Estimation Models



Technical Software Metrics

- A high level architectural design metric proposed by Henry and Kafura makes use of fan-in and fan-out in program architecture.
- Henry-Kafura-Metric (HKM) is calculated as

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

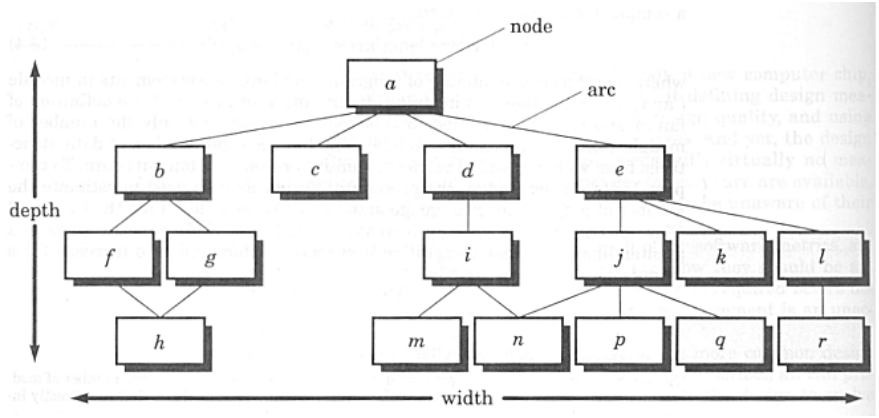
length(i) = the number of programming language statements in module i

f_{in} = fan-in for module i

f_{out} = fan-out for module i

Technical Software Metrics Example

- If length(i) is 100 programming language statements for module ***e***, HKM for module ***e*** would be equal to 400, since fan-in and fan-out for module ***e*** is 1 and 3 respectively.



Size & Arc-to-node Ration

- Fenton proposes other metrics in relation to program architecture:
  - **Size** = n (the number of nodes/module) + a (number of arcs/lines of control)

    = 17 + 18 = 35
- **Depth** = the longest path from the root (top) node to a leaf node. (Depth = 4)
- **Width** = maximum number of nodes at any one level of the architecture. (Width = 6)
- **Arc-to-node Ration**, r = a/n

  = 18/17 = 1.06

Software Maturity Index

$M_T =$ 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 in the current release

- The software maturity index is then computed in the following manner:

$$SMI = \frac{[M_T - (F_a + F_c + F_d)]}{M_T}$$

Metrics for Small Organizations

- The small projects and small organizations can also benefit economically from the intelligent use of software metrics.
- The key is to select metrics to compute carefully and that the data collection process is not too burdensome for the software developers.

Formulation Principles

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

Collection and Analysis Principles

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

Attributes for the Matrix

- **Simple and Computable** – Easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- **Empirically and Intuitively Persuasive** – Satisfy the engineer's intuitive notions about the product attribute under consideration
- **Consistent and Objective** – Results that are unambiguous.
- **Consistent in its Use of Units and Dimensions** – Use measures that do not lead to bizarre combinations of unit.
- **Programming Language Independent** – Based on the analysis model, the design model, or the structure of the program itself.
- **Effective Mechanism for Quality Feedback** – Provide a software engineer with information that can lead to a higher quality end product