

# ELEC 876: Software Reengineering (Software Metrics and Quality)

**Dr. Ying Zou**

Department of Electrical & Computer Engineering  
Queen's University



## Why Measure Software?

- Estimate cost and effort
  - Measure correlation between specifications and final product
- Improve productivity
  - Measure value and cost of software
- Improve software quality
  - Measure usability, efficiency, maintainability
- Improve reliability
  - Measure mean time to failure, etc.
- Evaluate methods and tools
  - Measure productivity, quality, reliability ...

***“You cannot control what you cannot measure”***

***– Tom DeMarco***

## What are Software Metrics?

- A software metric is a technique or method that applies software measurements to a class of software engineering objects to achieve a predefined goal
- Characteristics of software metrics:
  - Object of measurement
    - Products, processes and projects
  - Purpose of measurement
    - Characterization, assessment, evaluation, prediction
  - Source of measurement
  - Measured property
  - Context of measurement

## Software Metrics Categories

- Product metrics
  - Used to measure aspects of artifacts delivered to the customer (the internal and external characteristics of a system)
  - Information flow metrics, function points, complexity
- Process metrics
  - Used to measure aspects of the development and maintenance process
  - example: time to correct defect, number of components changed per correction
- Project metrics
  - Used to measure project management

# External Product Attributes

- External Product Attributes
  - **Definition:** measures how the product behaves in its environment
  - example: number of system defects perceived, time to learn the system
- Pros and Cons
  - Advantages:
    - close relationship with quality factors or goals
  - Disadvantages:
    - measure only after the product is used or process took place
    - data collection is difficult often involves human intervention/interpretation
    - relating external effect to internal cause is difficult

# Internal Product Attributes

- Internal Product Attribute
  - **Definition:** is measured purely in term of the product, separate from behaviours
  - example: method size, class coupling and cohesion
- Pros and Cons
  - advantages:
    - can be measured at any time
    - data collection is quite easy and can be automated
    - direct relationship between measured attribute and cause
  - disadvantage:
    - relationship with quality factors is not empirically validated
    - measurements may only be used as indicators, i.e. a heuristic

# Goal-Question-Metric (GQM) Approach

- Define Goal
  - E.g. “Reduce maintenance costs by 50% within one year”
- Break down into questions
  - How much do we spend on maintenance each month?
  - What fraction of our maintenance costs do we spend on each application we support?
  - How much money do we spend on adaptive, perfective, and corrective maintenance?
- Pick suitable metrics
  - Simple data you can count directly
    - Such as, total budget spend on maintenance
  - Metrics computed from two or more data items
    - Such as hours spent on each of the three maintenance activity types and the total maintenance cost over a period of time

## Metrics Assumptions

- Assumptions
  - A software property can be measured
  - The relationship exists between what we can measure and what we want to know
  - This relationship has been formalized and validated
- It may be difficult to relate what can be measured to desirable quality attributes
- Measurement analysis
  - Not always obvious what data means
    - Analyzing collected data is very difficult
  - Professional statisticians should be consulted if available
  - Data analysis must take local circumstances into account

## Software Product Metrics

- Axiom: Good internal structure implies good external quality
- Metrics provide a way to measure different features of software systems
- We will briefly discuss the following metrics
  - Software size & length related metrics
  - Metrics that relate with amount of delivered functionality
  - Metrics related to control flow complexity
  - Metrics that relate to reuse
  - Metrics that relate to development effort

## Lines of Code

- Length of a software system is measured in terms of lines of code (*LOC*)
- Lines of Code can be classified as lines of code that are comments or just blank lines (*CLOC*), and lines of code that contain executable source statements (*NCLOC*). Therefore

$$LOC = NCLOC + CLOC$$

- A useful metric is the Density of Comments defined as:

$$CLOC/LOC$$

```

1 public void paint ( Graphics g )
2 {
3 print (g, "Sequence in original order ", a, 25, 25);
4 sort();
5 print(g, "results", a,25, 55);
6 }
7 public void sort()
8 {
9 for (int pass=1; pass <a.length; pass++)
10 for (int i=0; i<a.length; i++)
11 if (a[ i ] > a[ i+1] )
12 { hold = a[i];
13 a[i] = a[i+1];
14 a[i+1]=hold;
15 }
16 }

```

**Figure 13.2** Java bubble sort.

```

1 public void paint ( Graphics g )
2 {
3 print (g, "Sequence in original order ", a, 25, 25); sort(); print(g, "results", a, 25, 55);
4 }
5 public void sort()
6 {
7 for (int pass=1; pass <a.length; pass++)
8 for (int i=0; i<a.length; i++) if (a[ i ] > a[ i+1] )
9 { hold = a[i]; a[i] = a[i+1]; a[i+1]=hold; }
10 }

```

**Figure 13.3** Shortened bubble sort code.

## Functionality Related Metrics

- The functionality of a product originates from an intuitive notion of the amount of function it delivers
- One of the most applied metrics in this category is the Function Point metric (FP)
- To compute the FP metric we have to compute first the ***Unadjusted Function Count*** metric and the ***Technical Complexity Factor***. Then

$$FP = UFC \times TCF$$

## Functionality Related Metrics (con't)

- To compute the UFC we examine:
  - Number of External Inputs
  - Number of External Outputs
  - Number of External Inquiries
  - Number of External Interface Files
  - Number of Internal Logic Files

UFC is

$$UFC = \sum_{i=1}^n (Item_i) \times weight_i$$

## Function Points (con't)

- Factors  $F_1, \dots, F_{14}$  that contribute to TCF are the number of modules that provides:

- |                                  |                        |
|----------------------------------|------------------------|
| 1. Reliable back-up and recovery | 7. Multiple sites      |
| 2. Distributed functions         | 8. Data communications |
| 3. Heavily used configuration    | 9. Performance         |
| 4. Operational ease              | 10. On-line data entry |
| 5. Complexity interface          | 11. On-line update     |
| 6. Reusability                   | 12. Complex processing |
|                                  | 13. Installation ease  |
|                                  | 14. Facilitate changes |

- TCF then is computed as:

$$TCF = 0.65 + 0.01 \times \sum_{i=1}^{14} F_i$$

# McCabe's Cyclomatic number

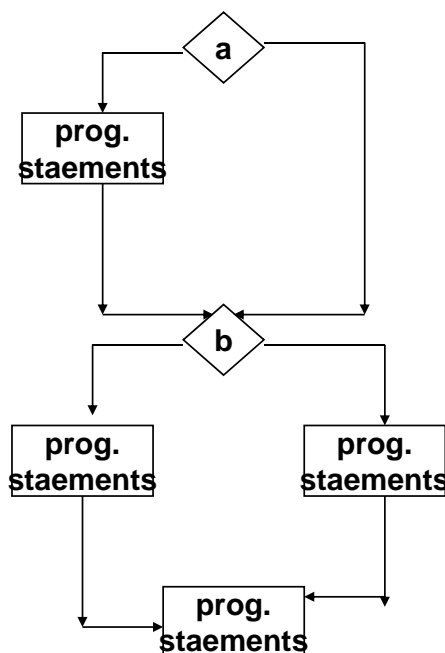
- A popular metric used to describe the complexity of a program was introduced by T.J. McCabe in 1976 and it is called the cyclomatic number.
- Cyclomatic number is based on the control flow of a program as follows:
  - $C = e - n + 2 * p$ 
    - Where e = number of edges
    - Where n = number of nodes
    - Where p = number of disconnected parts (usually it is 1)
  - in case where the control graph has only binary decisions, then  $C = b + 1$ 
    - Where b = number of binary decisions

Fall 2004, 2005

Elec 876: Software Reengineering

15

## Cyclomatic Example



# of edges = 7  
# of nodes = 6  
p = 1

$C = e - n + 2p$   
 $C = 7 - 6 + 2 = 3$

# binary decision = 2

$C = b + 1$   
 $C = 2 + 1 = 3$

Fall 2004, 2005

Elec 876: Software Reengineering

16

## Reuse Metrics Related Metrics

- Reuse is measured as public reuse and private reuse. The formula are:

$$\mathbf{Public\ Reuse = P_2/P_1}$$

Where  $P_1$  is the new written code, and  $P_2$  the externally obtained code and

***PrivateReuse = the extend to which modules within a product are reused within the same product***

## Development Effort Predicting Metrics

- One of the most popular metric for predicting development effort is the Henry-Kafura metric (Information Flow).

$$\mathbf{IF = (Fan-in(P) \times Fan-out(P))^2}$$

where

- Fan-in(P): denotes the number of local flows that terminate at module P plus the number of data structures from which information is retrieved by P
- Fan-out(P): denotes the number of local flows that emanate from module P plus the number of data structures updated by module P

## Example of Henry-Kafura metric

- **Consider 3 modules in a software component with the following fan-in and fan-out:**
  - module a : 3 fan-ins and 5 fan-outs
  - module b : 2 fan-ins and 1 fan-out
  - module c : 4 fan-ins and 2 fan-outs
- **Assume the weight is a uniformly 1 in this case.**
  - $whk_a = (3 * 5)^2 = 225$
  - $whk_b = (2 * 1)^2 = 4$
  - $whk_c = (4 * 2)^2 = 64$
- **The  $HK = 225 + 4 + 64 = 293$**

## Development Effort Predicting Metrics (con't)

- **Notes:**
  - Recursive module calls are treated as normal calls
  - Any variable shared by two or more modules is considered global to these modules
  - Compiler and Library modules are ignored
  - One level indirection for local flow (no tracing of calls)
  - No dynamic analysis considered
  - Duplicate flows are ignored
  - No module length related metrics are considered

# Metrics in Reengineering

- Estimating Cost
  - Is it worthwhile to reengineer, or is it better to start from scratch?
- Assessing Software Quality
  - Which components have poor quality? (Hence should be reengineered)
  - Which components have good quality? (Hence should be reverse engineered)
    - Metrics as a reengineering tool!
- Controlling the Reengineering Process
  - Trend analysis: which components did change?
  - Which refactorings have been applied?
    - Metrics as a reverse engineering tool!

# Coupling Metrics

- Coupling: The level of interconnection and dependency between modules. We aim for low coupling. The levels of coupling are:
  - Content Coupling – High Coupling ( $P_5(p, q)$ ): If  $p$  refers to the inside of  $q$  (i.e., branches into, changes data, alters a statement in  $q$ )
  - Common Coupling ( $P_4(p, q)$ ): if  $p$  and  $q$  refer to the same data
  - Control Coupling ( $P_3(p, q)$ ): If  $p$  passes a parameter to  $q$  with the intention of controlling its behavior
  - Stamp Coupling ( $P_2(p, q)$ ):  $p, q$  both accept the same record type as parameter. This type of coupling may manufacture interdependencies between otherwise unrelated modules
  - Data Coupling – Low Coupling ( $P_1(p, q)$ ):  $p, q$  communicate by parameters each one being either a single data element or an homogeneous set of data items which do not incorporate any control element

## Coupling Metrics (con't)

- The overall coupling value between two modules is given as

$$c(p, q) = i + \frac{n}{n + 1}$$

where  $i$ : is the strongest coupling type between  $p$  and  $q$

$n$  is the number of interconnections between  $p$  and  $q$

- The overall coupling for a module is given as:

$$C(D_i) = \text{median value of } \{c(D_i, D_j)\} \text{ } j \neq i$$

## Line of Code (con't)

- A metric for predicting final system's length is :

$$LOC = \alpha \times \sum_{i=1}^n S_i$$

Where

–  $S_i$  is the size of module  $i$  in terms of components of its design

–  $\alpha$  is a design to code expansion ratio defined as:

$$\text{Size Of the Design} / \text{Size of the Code}$$

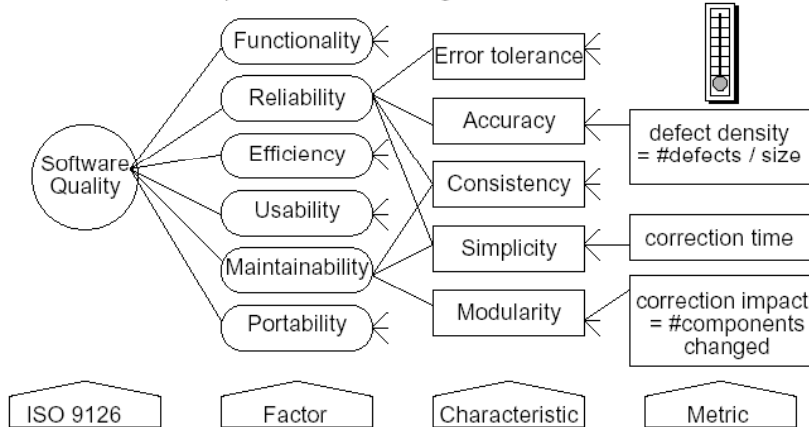
Empirical measurements show that size of the design  $D$  relates to size of the code  $L$  with the following formula

$$D = 49 \times L^{1.01}$$

–  $n$  is the number of modules in the design

# Quantitative Quality Model

- Quality according to ISO 9126 standard
  - Divide-and conquer approach via “hierarchical quality model”
  - Leaves are simple metrics, measuring basic attributes



# Define Quality Model

- Choose the characteristics, design principles, metrics, and the thresholds

