CSE470:Software Engineering

video lecture series produced by:

A.M.Esfar-E-Alam

Afrina Khatun

Dr.Muhammad Zavid Parvez

Hossain Arif

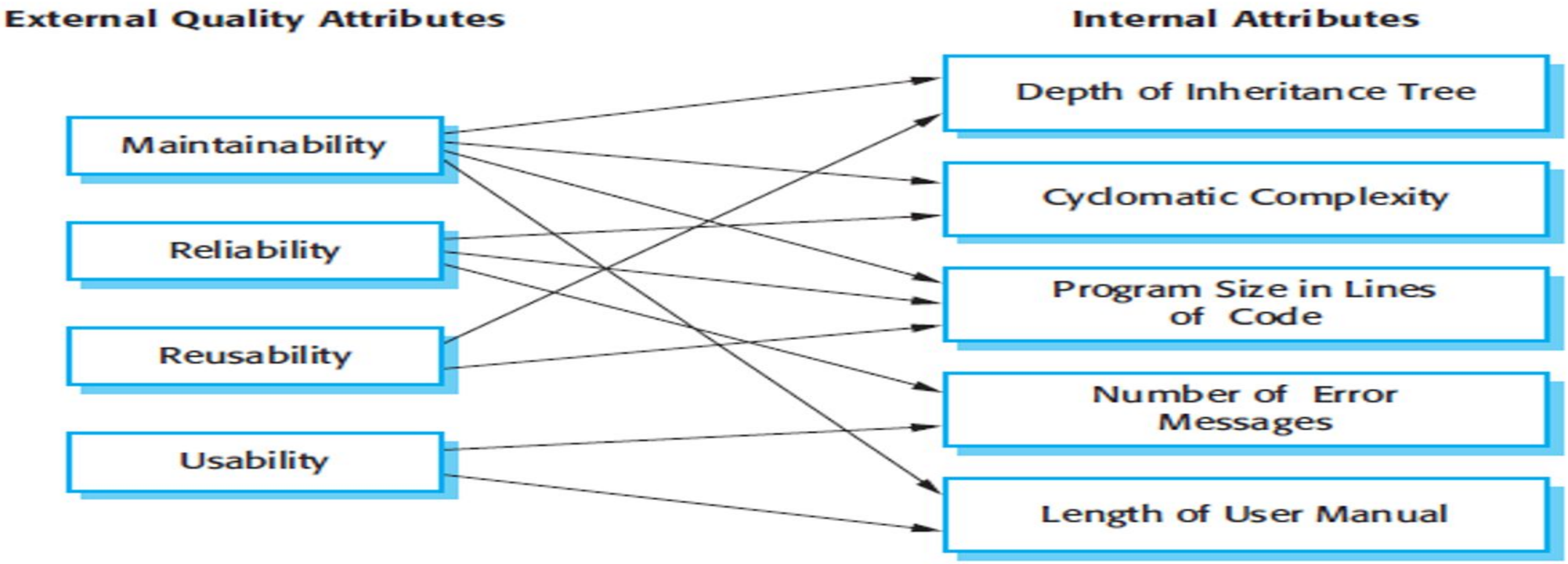
Software measurement and metrics

- ➔ Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- ➔ This allows for objective comparisons between techniques and processes.
- ➔ Although some companies have introduced measurement programmes, most organisations still don’t make systematic use of software measurement.
- ➔ There are few established standards in this area

Software metric

- ➔ Any type of measurement which relates to a software system, process or related documentation
- ➔ **Lines of code in a program**, the **Fog index**, **number of person-days** required to develop a component.
- ➔ Allow the software and the software process to be quantified.
- ➔ May be used to predict product attributes or to control the software process.
- ➔ Product metrics can be used for general predictions or to identify anomalous components.

Relationships between internal and external software



Product metrics

- A quality metric should be a predictor of product quality.
- Classes of product metric
- **Dynamic metrics** which are collected by measurements made of a program in execution;
- **Static metrics** which are collected by measurements made of the system representations;
- Dynamic metrics help assess efficiency and reliability
- Static metrics help assess complexity, understandability and maintainability.

Comparison of Dynamic vs Static Metrics in Software Engineering

Characteristic	Dynamic Metrics	Static Metrics
Measurement Context	Measured during software execution (runtime).	Measured from the source code or design documents.
Focus	Performance, behavior, resource utilization, runtime attributes.	Code structure, complexity, design quality.
Tools for Measurement	Profilers, performance monitors, logging systems.	Static analysis tools, code quality checkers.
Example Metrics	CPU usage, memory usage, execution time, error rate, throughput.	Lines of code (LOC), cyclomatic complexity, code duplication, class count.
Nature of Metric	Change over time depending on conditions.	Fixed unless the codebase changes.

Fan-in/Fan-out, Length of code

- Fan-in/Fan-out
  - Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
- Length of code
  - This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the **most reliable metrics** for predicting error-proneness in components

Fan-in/Fan-out Example

```
def process_payment(payment_info):
    validate_payment(payment_info)         # Fan-out 1
    calculate_tax(payment_info)            # Fan-out 2
    check_fraud(payment_info)              # Fan-out 3
    send_confirmation_email(payment_info) # Fan-out 4
    update_transaction_record(payment_info) # Fan-out 5
    process_credit(payment_info)           # Fan-out 6
```

```
# Logging Utility Module
def log_error(error_message):
    # Logs the error message to a file or
monitoring system.
    pass
```

```
def process_payment(payment_info):
   if not is_valid(payment_info):
      log_error("Invalid payment info")  # Fan-in 1
      return

def process_order(order_info):
   if not is_available(order_info):
      log_error("Out of stock")  # Fan-in 2
      return

def user_login(username, password):
   if not is_authenticated(username, password):
      log_error("Failed login attempt")  # Fan-in 3
      return
```

CYCLOMATIC COMPLEXITY

- ➔ Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors.
- ➔ It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.
- ➔ Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand. It can be represented using the below formula:

M = **E** − **N** + **2P**,where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

The complexity M is then defined as

$$\mathbf{M = R + 1},$$

where R = the number of regions in the graph.

The complexity M is then defined as

$$\mathbf{M = P + 1},$$

where P = the number of predicate nodes in the graph.

These two formulas are easy to use

SPECIALIZATION INDEX (SIX)

- ➔ The **Specialization Index metric** measures the extent to which subclasses override their ancestors classes. This **index** is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class
- ➔ The metric provides a percentage, where the class contains at least one operation. For a root class, the specialization indicator is zero. Nominal range is between 0 % and 120 %.

The ... variable	represents the ...
DIT	depth of inheritance
NMA	the number of operations added to the inheritance
NMI	the number of inherited operations
NMO	the number of overloaded operations

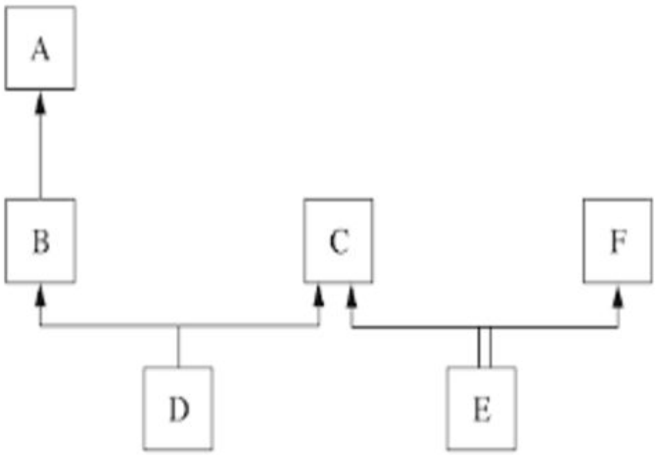
NMO – Number of Overridden Methods not Overloaded.

Example:

$$SIX = \frac{3 \times 4}{3 + 4 + 3} \times 100 = 120$$

DIT(D) = 2

DIT(E) = 1



```
Class Person{

 void read();

 void display();

}

Class Student extends Person{

 void read();

void display();

Void getAverage();

}

Class GraduateStudent extends Student{

 void read();

void display();

Void workStatus();

}
```

The … variable	represents the …
DIT	depth of inheritance
NMA	the number of operations added to the inheritance
NMI	the number of inherited operations
NMO	the number of overloaded operations

$$SIX = \frac{NMO \times DIT}{NMO+NMA+NMI}$$

$$SIX = \frac{2 \times 2}{2 + 1 + 1}$$

-> 100% [(4/4)*100]

DEFECT REMOVAL EFFICIENCY

- *A **defect** is found when the application does not conform to the requirement specification.*
- *A mistake in coding is called **Error***
- An average DRE score is usually around 85% across a full testing program.
- *DRE = E / (E + D) where:*
- *E* is the number of errors found before delivery of the software to the end-user
- *D* is the number of defects found after delivery.
- We found 100 defects during the testing phase and then later, say within 90 days after software release (in production), found five defects,
- DRE = 100/(100+5) = 95.2%