

## COURSE NAME

OBJECT ORIENTED  
ANALYSIS AND DESIGN

CSC 2210

(UNDERGRADUATE)



---

## CHAPTER 9

## SOFTWARE PROJECT ESTIMATION

---

VICTOR STANY ROZARIO

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE, AIUB

Web: <https://cs.aiub.edu/profile/stany>

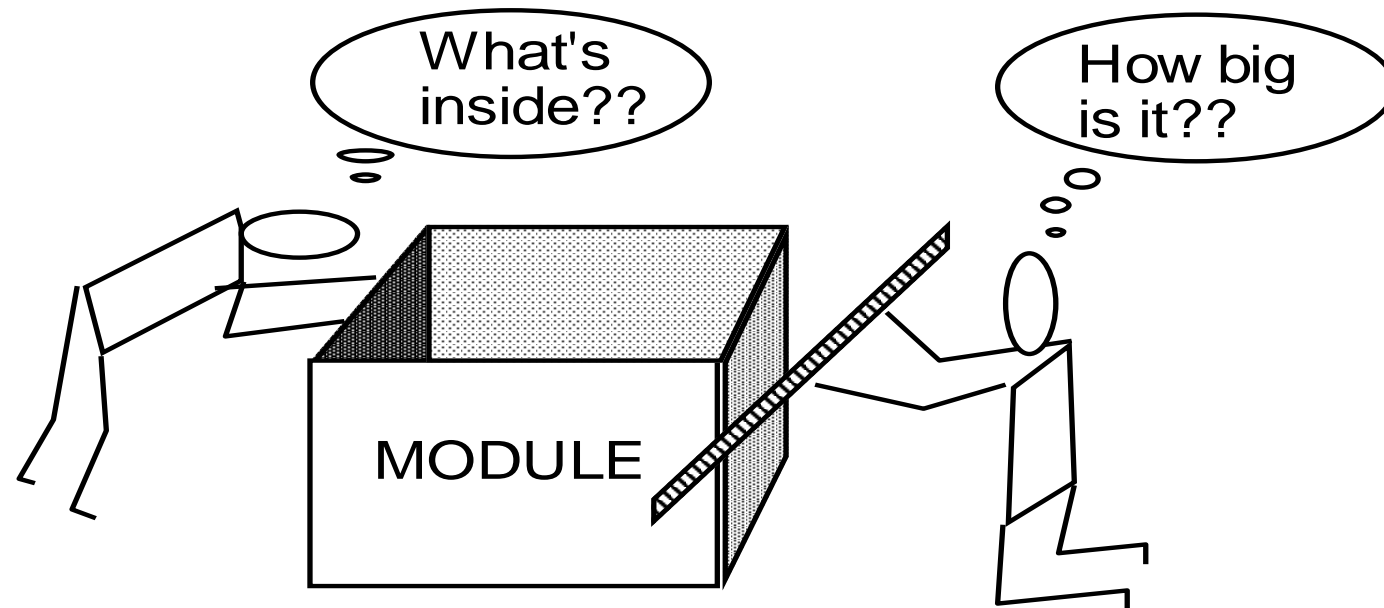


# SOFTWARE MEASUREMENT

- Software measurement is a **quantified attribute of a characteristic** of a software product or the software process (e.g., **quality, complexity**, etc.)
- The ability to measure attributes of software and the software development process are essential to improvements in the practice of software engineering
- Measurements enables us to evaluate the situation properly (**objective evaluation**).
- It can also be applied to **improve the software process** and hence **assist project management**

# SOFTWARE METRICS

- The common measurement entities are the **software metrics**, which helps us to evaluate: **Performance, Efficiency, Security, Complexity, Testability, Flexibility**, etc.
- **Metric is a quantifiable measure** (characteristic) of software, e.g., **2 errors were discovered by customers in 18 months** (more meaningful than saying that **2 errors were found**)



## SOFTWARE METRICS : SIZE

- Measuring the size of a program is by counting the number of lines of code (LOC)
- During maintenance, the focus is on number of lines of code that have been added or modified during a maintenance process.
- Easy to determine and also correlates strongly with other measures such as effort and error density.

## SOFTWARE METRICS : COMPLEXITY

- One of the major problems that software maintainers face is dealing with the increasing complexity of the source code that they have to modify (Cyclomatic code complexity).
- Includes program structure, semantic content, control flow, data flow, and algorithmic complexity.
- The more complex a program is, the more likely it is for the maintainer to make an error when implementing a change.
- The higher the complexity value, the more difficult it is to understand the program, hence making it less maintainable.

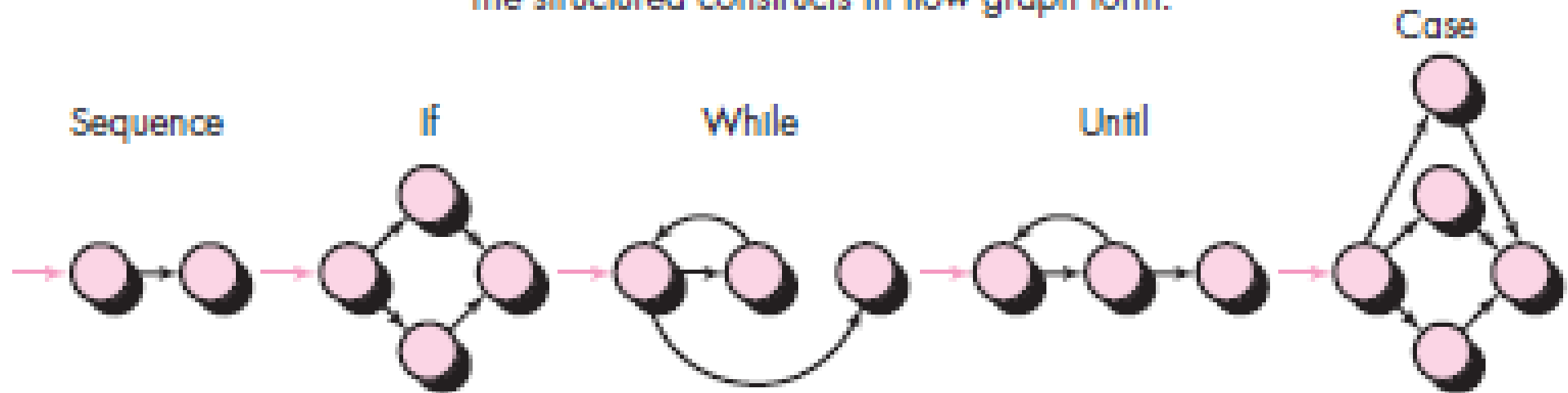
## SOFTWARE MEASURES

- ☐ Cyclomatic code complexity (CCC)
- ☐ Weighted Methods per Class (WMC)
- ☐ Depth of Inheritance Tree (DIT)
- ☐ Number of Children (NOC)
- ☐ Coupling between Classes (CBC)
- ☐ Lack of Cohesion in Methods (LCOM)
- ☐ COCOMO

## MCCABE'S CYCLOMATIC COMPLEXITY

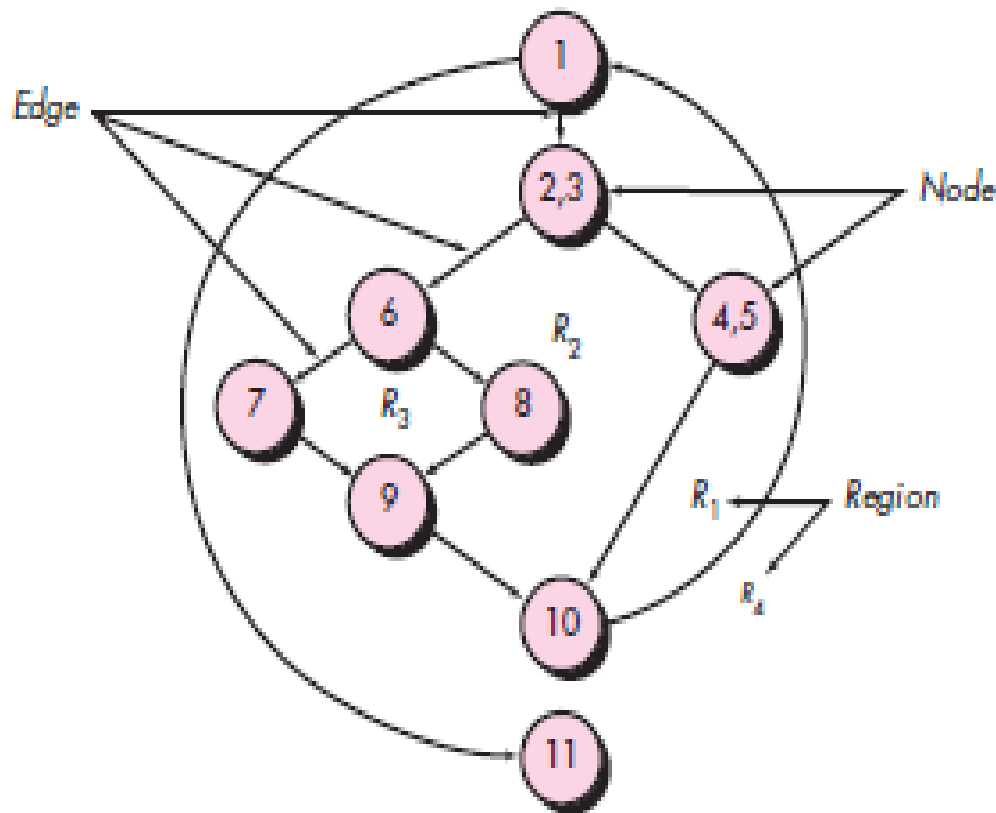
- McCabe views a program as a **directed graph** in which **lines of program statements are represented by nodes** and **the flow of control between the statements is represented by the edges.**

The structured constructs in flow graph form:



Where each circle represents one or more nonbranching PDL or source code statements

# INDEPENDENT PROGRAM PATHS



Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

- Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- How do you know how many paths to look for? The computation of **cyclomatic complexity** provides the answer.



## MCCABE'S CYCLOMATIC COMPLEXITY

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. Complexity is computed in one of four ways:

1. The number of independent path.
2. The number of regions of the flow graph corresponds to the cyclomatic complexity.  
(in the previous example = 4)
3. Cyclomatic complexity  $V(G)$  for a flow graph  $G$  is defined as  $V(G) = E - N + 2$   
(in the previous example  $11 - 9 + 2 = 4$ )
  - where  $E$  is the number of flow graph edges and  $N$  is the number of flow graph nodes.
4. Cyclomatic complexity  $V(G)$  for a flow graph  $G$  is also defined as  $V(G) = P + 1$   
(in the previous example  $3 + 1 = 4$ ) [condition: 1 - 2, 3 - 6]
  - where  $P$  is the number of predicate nodes (containing a condition) contained in the flow graph  $G$ .

## MCCABE'S CYCLOMATIC COMPLEXITY

- ❑ Used as an indicator of the psychological complexity of a program.
- ❑ During maintenance, a program with a very high cyclomatic number (usually above 10) is considered to be very complex.
- ❑ It helps to identify highly complex programs that may need to be modified in order to reduce complexity.
- ❑ The cyclomatic number can be used as an estimate of the amount of time required to understand and modify a program.
- ❑ The flow graph generated can be used to identify the possible test paths during testing.

## MCCABE'S CYCLOMATIC COMPLEXITY

- ❑ It takes no account of the complexity of the conditions in a program
- ❑ In its original form, it failed to take account of the degree of nesting in a program.

Cyclomatic complexity Threshold Values	Risk evaluation
01 – 10	Simple Program, without much risk
11 – 20	More complex program, moderate risk
21 – 50	Complex program, high risk
> 50	Un-testable program, very high risk

## WEIGHTED METHODS PER CLASS (WMC)

- ❑ The effort in developing a class will in some sense will be determined by the number of methods the class has and the complexity of the methods.
- ❑ Suppose a class  $C$  has methods  $M1, M2... Mn$  defined on it. Let the complexity of the method  $M1$  be  $c1$ , then

$$WMC = \sum_{i=1}^{i=n} c_i$$

- ❑ If the complexity of each method is considered 1, WMC will be (gives) the total number of methods in the class.

## WEIGHTED METHODS PER CLASS (WMC)

- ❑ The data based on evaluation of some existing programs, shows that in most cases, the classes tend to have only a small number of methods, implying that most classes are simple and provide some specific operations
- ❑ WMC metric has a reasonable correlation with fault-proneness of a class. As can be expected, the larger the WMC of a class the better the chances that the class is fault-prone

Number of Complex Methods in a class	Effect
High	<ul style="list-style-type: none"><li>- Potentially Error Prone</li><li>- Impact on the children of the class</li><li>- Reusability decreases</li></ul>
Low	<ul style="list-style-type: none"><li>- Reusability Increases</li></ul>

## DEPTH OF INHERITANCE TREE (DIT)

- ❑ The DIT of a class  $C$  in an inheritance hierarchy is the depth from the root class in the inheritance tree
- ❑ Also, one of the main mechanisms for reuse
- ❑ The deeper a particular class is in a class hierarchy, the more methods it has available for reuse, thereby providing a larger reuse potential
- ❑ Inheritance increases **coupling**, which makes changing a class harder
- ❑ A class deep in the hierarchy has a lot of methods it can inherit, which makes it difficult to predict its behavior

## DEPTH OF INHERITANCE TREE (DIT)

- ❑ It is the length of the path from the root of the tree to the node representing  $C$  or the number of **ancestors**  $C$  has
- ❑ In case of **multiple inheritance**, the **DIT metric is the maximum length from a root to  $C$**
- ❑ Statistical data suggests that most classes in applications tend to be close to the root, with the maximum DIT metric value being around 10
- ❑ **Most the classes have a DIT of 0** (that is, they are the root)
- ❑ **The designers** tend to **keep** the number of inheritance levels in the **inheritance tree small**, presumably to aid understanding. In other words, designers might be **giving up on reusability in favor of comprehensibility**

## DEPTH OF INHERITANCE TREE (DIT)

- The experiments show that DIT is very significant in predicting defect-proneness of a class: the higher the DIT the higher is the probability that the class is defect-prone

Attribute	Increase ( ↑ ) / Decrease ( ↓ ) Along with increasing DIT
Maintainability	↓
Portability	↓
Functionality	↑
Efficiency	↓

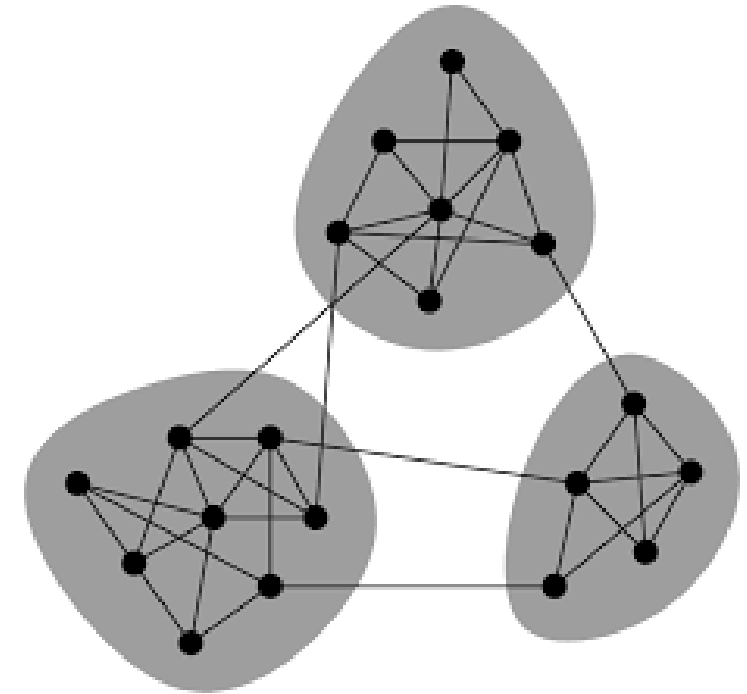


## NUMBER OF CHILDREN (NOC)

- ❑ The number of children (NOC) metric value of a class  $C$  is the number of **immediate subclasses** of  $C$
- ❑ This metric can be used to evaluate the degree of reuse, as a higher NOC number reflects reuse of the definitions in the superclass by a larger number of subclasses
- ❑ It also gives an idea of the direct influence of a class on other elements of a design
- ❑ The larger the influence of a class, the more important the class is correctly designed
- ❑ In the empirical observations, it was found that classes generally had a small NOC metric value, with a vast majority of classes having no children
- ❑ This suggests that in the systems analyzed, inheritance was not used very heavily
- ❑ The data suggest that the larger the NOC, the lower the probability of detecting defects in a class

## COUPLING BETWEEN CLASSES (CBC)

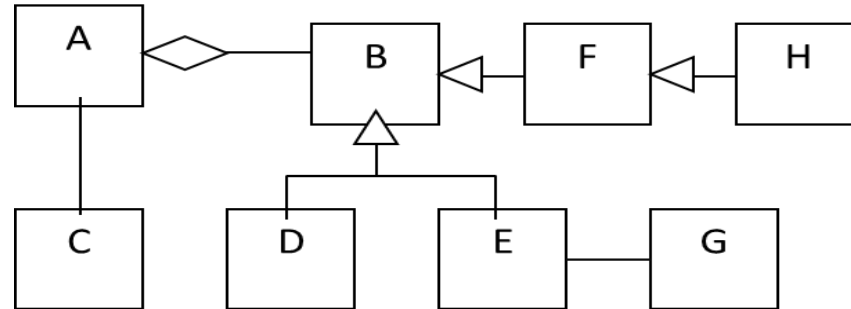
- ❑ Coupling between classes of a system reduces modularity and make class modification harder
- ❑ It is desirable to reduce the coupling between classes
- ❑ The less coupling of a class with other classes, the more independent the class, and more easily modifiable
- ❑ Coupling between classes (CBC) is a metric that tries to quantify coupling that exists between classes
- ❑ The CBC value for a class  $C$  is the total number of other classes to which the class is coupled
- ❑ Two classes are considered coupled if methods of one class use methods or instance variables defined in the other class (inheritance)



## COUPLING BETWEEN CLASSES (CBC)

- ❑ There are indirect forms of coupling (through pointers, etc.) that are hard to identify
- ❑ The experimental data indicates that most of the classes are self-contained and have low CBC value
- ❑ Some types of classes, for example, the ones that deal with managing interfaces, generally tend to have higher CBC values
- ❑ The data found that CBC is significant in predicting the fault-proneness of classes, particularly those that deal with user interfaces

## EXAMPLE (NOC, DIT, CBC)



CLASS	NOC	Influence on Design	DIT	Reuse Potential	CBC
A	0	Low	0	Low	Low
B	3	Highest	0	Low	Highest
C	0	Low	0	Low	Lowest
D	0	Low	1	Moderate	Low
E	0	Low	1	Moderate	Moderate
F	1	Moderate	1	Moderate	Moderate
G	0	Low	0	Low	Lowest
H	0	Low	2	Highest	Lowest

## LACK OF COHESION IN METHODS (LCOM)

- ❑ Cohesion captures how closely bound the different methods of the class are
- ❑ Two methods of a class  $C$  can be considered “cohesive” if the set of instance variables (class variable) of  $C$  that they access, have some elements in common
- ❑ Let  $I_i$  and  $I_j$  be the set of instance variables accessed by the methods  $M_i$  and  $M_j$ ,  $Q$  be the set of all cohesive pairs of methods, that is, all  $(M_i, M_j)$  such that  $I_i$  and  $I_j$  have a non-null intersection. Let  $P$  be the set of all non-cohesive pairs of methods, that is, pairs such that the intersection of sets of instance variables they access is null. Then LCOM is defined as

$$\text{LCOM} = |P| - |Q|, \text{ if } |P| > |Q|, \text{ otherwise } 0$$

- ❑ Coupling (increase)  $\rightarrow$  Cohesion (decrease)  $\rightarrow$  LCOM (increase)

## EXAMPLE (LCOM)

CLASS A
a1 a2 a3 a4
A1(a1, a2) A2(a1) A3 (a4) A4 (a1, a4)

$LCOM = |P| - |Q|$ , if  $|P| > |Q|$ , otherwise 0

Pairs:

(A1, A2), (A1, A3), (A1, A4), (A2, A3), (A2, A4),  
(A3, A4)

P = 2 (Non-Cohesive pairs)

Q = 4 (Cohesive pairs)

$Q > P$

$LCOM = 0$

## LACK OF COHESION IN METHODS (LCOM)

- ❑ If there are  $n$  methods in a class  $C$ , then there are  $n(n-1)$  pairs, and LCOM is the number of pairs that are non-cohesive minus the number of pairs that are cohesive
- ❑ The larger the number of cohesive methods, the more cohesive the class will be, and the LCOM metric will be lower
- ❑ A high LCOM value may indicate that the methods are trying to do different things and operate on different data entities
- ❑ If this is validated, the class can be partitioned into different classes
- ❑ High cohesion & low LCOM is a highly desirable property for modularity.
- ❑ There is little significance of this metric in predicting the fault-proneness of a class

## COCOMO (CONSTRUCTIVE COST MODEL)

Based on SLOC characteristic, and operates according to the following equations:

- $\text{Effort} = \text{PM} = \text{Coefficient}_{\langle \text{Effort Factor} \rangle} * (\text{SLOC} / 1000)^P$  [100,000 SLOC/1000 = 100k SLOC]
- $\text{Development time} = \text{DM} = 2.50 * (\text{PM})^T$
- $\text{Required number of people} = \text{ST} = \text{PM} / \text{DM}$

**PM** : person-months needed for project (labor working hours)

**SLOC** : source lines of code

**P** : project complexity (1.04-1.24)

**DM** : duration time in months for project (weekdays)

**T** : SLOC-dependent coefficient (0.32-0.38)

**ST** : average staffing necessary

Software Project Type	Coefficient <Effort Factor>	P	T
Organic	2.4	1.05	0.38
Semi-detached	3.0	1.12	0.35
Embedded	3.6	1.20	0.32



## COCOMO (CONSTRUCTIVE COST MODEL)

### Organic:

- relatively small size (2-50 KLoc), simple, less innovative software projects in which a small teams with good application experience work to a project with less tightly schedule
- Example: showing VUES information to webpage, payroll system

### Semidetached:

- Medium size (50-300 KLoc) and innovation, complexity in software project in which teams with mixed experience levels works in a mix of h/w and s/w application with relative tight schedule
- Example: biometric log-in time saved in VUES database

### Embedded:

- A Large size (>300 KLoc), more innovative, complex software project that must be developed within a strongly coupled to hardware environment (e.g. Autopilot, biometric device, elevator)
- Projects has the characteristics that the product being developed had to operate within very tight constraints and changes to the system are very costly

## REFERENCES

- R.S. Pressman & Associates, Inc (2010). *Software Engineering: A Practitioner's Approach*.