ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Software Quality Assurance
Đảm bảo chất lượng phần mềm

Lecture 2: Software Metrics

Contents

2.1. Motivation of Software Metrics

How to achieve a successful software project?

- In order to conduct a successful software project we must understand
  - the scope of work to be done
  - the risks incurred
  - the resources required
  - the tasks to be accomplished
  - the milestones to be tracked
  - the cost
  - the schedule to be followed

Project Management

- Before a project can be planned
  - objectives and scope should be established
  - alternative solutions should be considered
  - technical and management constraints should be identified
- This information is required to estimate costs, tasks, and a schedule

Software Metrics

- Metrics help us understand the technical process that is used to develop a product
  - The process is measured to improve it and the product is measured to increase its quality
- Measuring software projects is still controversial
  - Not yet clear which are the appropriate metrics for a software project
  - whether people, processes or products can be compared using such metrics

Why use metrics?

- Without measurements there is no real way to determine if the process is improving
- They allow the establishment of meaningful goals for improvement
- Metrics allow an organization to identify the causes of defects that have the greatest effect on software development

Applying metrics

- When metrics are applied to a product they help identify
  - which user requirements are likely to change
  - which modules are most error prone
  - how much testing should be planned for each module

Direct measurements

- Measurements can be either direct or indirect
- Direct measures are taken from a feature of an item (e.g., length)
    - Lines of code, execution speed, memory size, defects reported

Indirect Measurements

- Indirect measurements associate a measure to a feature of the object being measured (e.g., quality is based upon counting rejects)
    - functionality, quality, complexity, efficiency, reliability and maintainability

Code Complexity Measurements
Các độ đo độ phức tạp mã nguồn

- Code complexity correlates with the defect rate and robustness of the application program
- Code complexity measurement tools
  - Testwell CMT++ for C, C++ and C#
  - Testwell CMTJava for Java
- Code with good complexity
  - contains less errors
  - is easier to understand
  - is easier to maintain

How to use code complexity measurements

- Code complexity metrics are used to locate complex code
- To obtain a high quality software with low cost of testing and maintenance, the code complexity should be measured as early as possible in coding --> developers can adapt their code when recommended values are exceeded

Code complexity metrics

- Metrics shown by Testwell CMT++/CMTJava
  - Lines of code metrics
  - McCabe Cyclomatic number
  - Halstead Metrics
  - Maintainability Index

2.2. Lines of code

Lines of code metrics

- Most traditional measures used to quantify software complexity
- Simple, easy to count and very easy to understand
- Do not take into account the intelligence content and the layout of the code
- Testwell CMT++/CMTJava calculates the following lines of code metrics
    - LOCphy: number of physical lines
    - LOCbl: number of blank lines (a blank line inside a comment block is considered to be a comment line)
    - LOCpro: number of program lines (declarations, definitions, directives and code)
    - LOCcom: number of comment lines

Lines of code metrics - Recommendations Những giá trị khuyên dùng cho chiều dài mã nguồn

- **<u>Function length</u>**
  - Function length should be 4 to 40 program lines
  - A function definition contains at least a prototype, one line of code, a pair of braces, which makes 4 lines
  - A function longer than 40 program lines probably implements many functions
  - Exception: functions containing one selection statement with many branches) --> decomposing them into smaller functions often decreases readability

Lines of code metrics - Recommendations
Những giá trị khuyên dùng cho chiều dài mã nguồn

- **File length**
  - File length should be 4 to 400 program lines
  - The smallest entity that may reasonably occupy a whole source file is a function and the minimum length of a function is 4 lines
  - Files longer than 400 program lines (10..40 functions) are usually too long to be understood as a whole

Lines of code metrics - Recommendations
Những giá trị khuyên dùng cho chiều dài mã nguồn

- **Comments**
  - At least 30% and at most 75% of a file should be comments
  - If less than one third of a file is comments the file is either very trivial or poorly explained
  - If more than 75% of a file are comments, the file is not a program but a document
  - Exception: in a well-documented header file percentage of comments may sometimes exceed 75%

2.3. Halstead's Metrics

Halstead's Metrics

- Developed by Maurice Halstead
- Introduced in 1977
- Used and experimented extensively since that time
- One of the oldest measures of program complexity
- Strong indicators of code complexity
- Often used as a maintenance metric

Halstead's Metrics (cont.)

- Based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand
- **Operator: toán tử – Operand: toán hạng**
- Then is counted
  - number of unique (distinct) operators (n1)
  - number of unique (distinct) operands (n2)
  - total number of operators (N1)
  - total number of operands (N2)
- All other Halstead measures are derived from these four quantities with certain fixed formulas as described later

Operators and Operands

- Operators
  - traditional: +, -, *, /, ++, --, etc.
  - keywords: return, if, continue, break, try, catch etc.
- Operands
  - identifiers
  - constants

Halstead’s Metrics Một số độ đo của Halstead

- **Program length (N):** the program length is the sum of the total number of operators and operands in the program:
  - $N = N_1 + N_2$
- **Vocabulary size (n):** the vocabulary size is the sum of the number of unique operators and operands
  - $n = n_1 + n_2$
- **Program volume (V):** information content of the program
  - $V = N * \log_2(n)$
- **Halstead's volume** describes the size of the implementation of an algorithm

Halstead's metrics - Recommendations Một số khuyến cáo về giá trị độ đo Halstead

- The volume of a function should be at least 20 and at most 1000
  - The volume of a parameter less one-line function that is not empty is about 20
  - A volume greater than 1000 tells that the function probably does too many things
  - The volume of a file should be at least 100 and at most 8000
  - These limits are based on volumes measured for files whose LOCpro and v(G) are near their recommended limits

Other Halstead's metrics

- **Difficulty level (D)**
  - The difficulty level or error proneness (D) of the program is proportional to the number of unique operator in the program
  - D is also proportional to the ration between the total number of operands and the number of unique operands
    - i.e., if the same operands are used many times in the program, it is more prone to errors
  - $D = (n1/2) * (N2 / n2)$
- **Program level (L)**
  - The program level (L) is the inverse of the error proneness of the program
  - i.e., A low level program is more prone to errors than a high level program
  - $L = 1 / D$

Other Halstead's metrics

- **Effort to implement (E)**
  - The effort to implement or understand a program is proportional to the volume and to the difficulty level of the program
  - E = V * D
- **Time to implement (T)**
  - The time to implement or understand a program (T) is proportional to the effort
  - Halstead has found that dividing the effort by 18 give an approximation for the time in seconds
  - T = E / 18

Other Halstead's metrics

- **Number of delivered bugs (B)**
  - The number of delivered bugs (B) correlates with the overall complexity of the software
  - B = (E^2/3)/3000
  - Estimates for the number of errors in the implementation
  - Delivered bugs in a file should be less than 2
  - Experiences have shown that when programming with C or C++, a source file almost always contains more errors than B suggests
  - B is an important metric for dynamic testing
  - The number of delivered bugs approximates the number of errors in a module
  - As a goal at least that many errors should be found from the module in its testing

Halstead’s metric Example
Ví dụ về độ đo Halstead

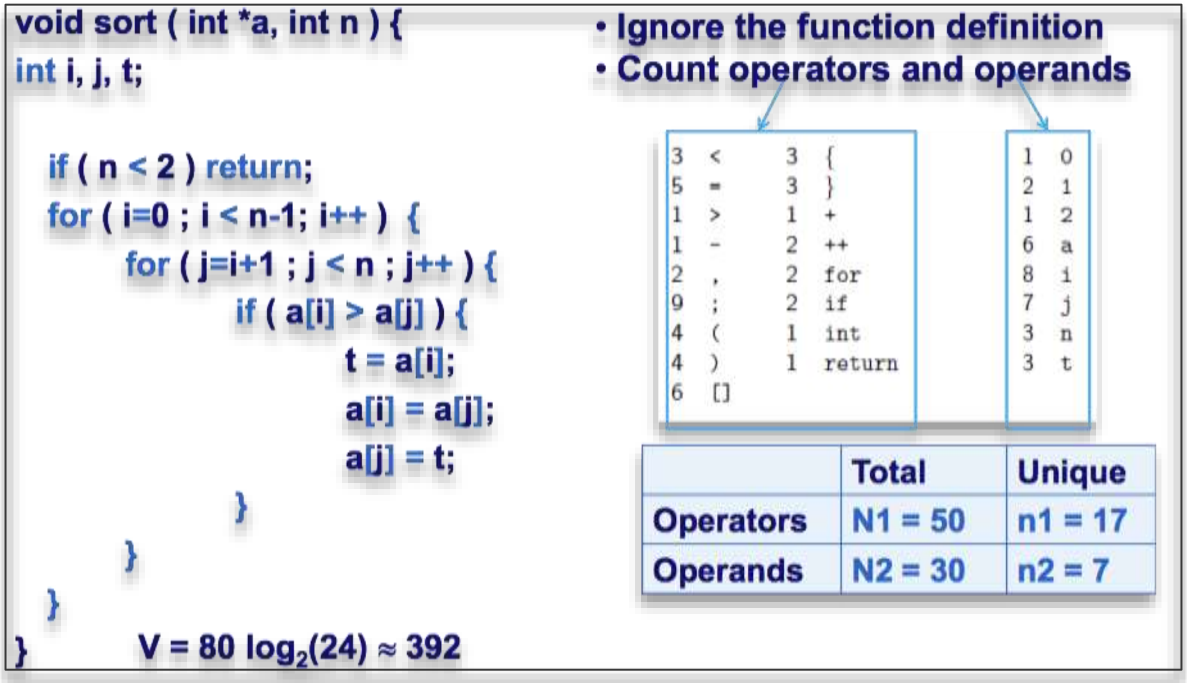
- Given a source code in C/C++
- Calculate some basic metrics of Halstead
    - n1
    - n2
    - N1
    - N2
- What are the operators/operands in this source code?

```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
          for ( j=i+1 ; j < n ; j++ ) {
                    if ( a[i] > a[j] ) {
                              t = a[i];
                              a[i] = a[j];
                              a[j] = t;
                    }
          }
   }
}
```

$V = 80 \log_2(24)$ □ 392

Halstead Example (cont.)

- Step 1:
  - List all the operands and operators in the source file
  - Count the number of distinct operands and distinct operators
- Step 2:
  - Calculate V, D, E…



```
void sort ( int *a, int n ) {
int i, j, t;

  if ( n < 2 ) return;
  for ( i=0 ; i < n-1; i++ )  {
       for ( j=i+1 ; j < n ; j++ ) {
              if ( a[i] > a[j] ) {
                     t = a[i];
                     a[i] = a[j];
                     a[j] = t;
              }
       }
  }
}
```

- Ignore the function definition
- Count operators and operands

```
3  <     3  {
5  =     3  }
1  >     1  +
1  -     2  ++
2  ,     2  for
9  ;     2  if
4  (     1  int
4  )     1  return
6  []
```

```
1  0
2  1
1  2
6  a
8  i
7  j
3  n
3  t
```

	Total	Unique
Operators	N1 = 50	n1 = 17
Operands	N2 = 30	n2 = 7

$V = 80 \log_2(24) \approx 392$

2.4. McCabe Cyclomatic Number

McCabe Cyclomatic Number
Chỉ số phức tạp của McCabe

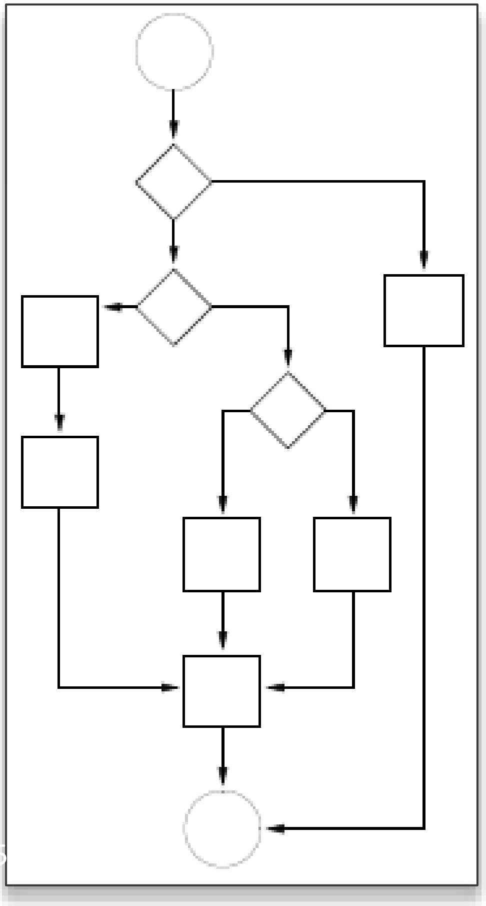
- The cyclomatic complexity v(G) has been introduced by Thomas McCabe in 1976
- Measures the number of linearly-independent paths through a program module (Control Flow)
- The McCabe complexity is one of the more widely-accepted software metrics, it is intended to be independent of language and language format
- Considered as a broad measure of soundness and confidence for a program

McCabe Cyclomatic Number – v(G)

- v(G) is the number of conditional branches
- v(G) = 1 for a program consisting of only sequential statements
- For a single function, v(G) is one less than the number of conditional branching points in the function
- The greater the cyclomatic number is, the more execution paths there are through the function, and the harder it is to understand

McCabe Cyclomatic Number
Công thức tính chỉ số McCabe

- Describes the structural complexity
  - Control flow
  - Data flow
- Graph based metrics
  - Number of vertices
  - Number of edges
  - Maximum length (depth of graph)
- In general: v(G) = #edges - #vertices + 2
- For control flow graphs
  - v(G) = #binaryDecision + 1
  - v(G) = #IFs + #LOOPs + 1



McCabe Cyclomatic Number
Ý nghĩa của chỉ số McCabe

- For dynamic testing, the cyclomatic number v(G) is one of the most important complexity measures
- Because the cyclomatic complexity describes the control flow complexity
    - it is obvious that modules and functions having high cyclomatic number need more test cases than modules having a lower cyclomatic number
    - Rule: each function should have at least as many test cases as indicated by its cyclomatic number

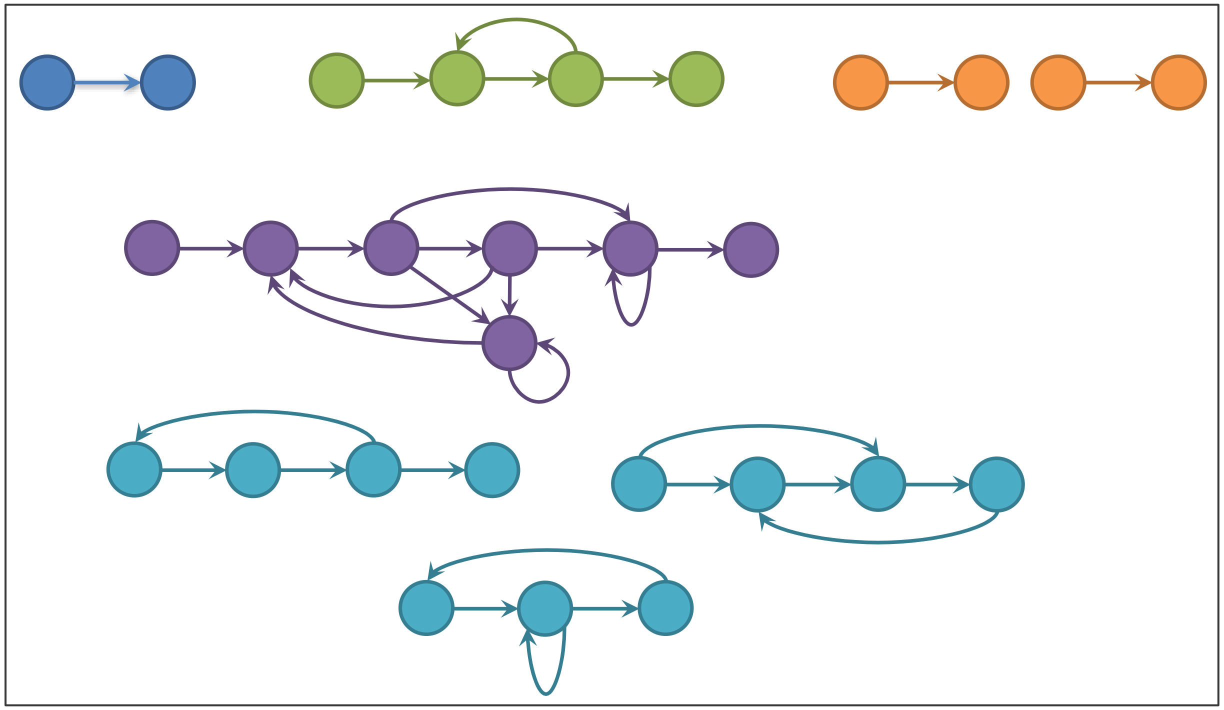
McCabe Cyclomatic Number - Example

- Count IFs and LOOPs
- IF: 2, LOOP: 2
- v(G) = 5

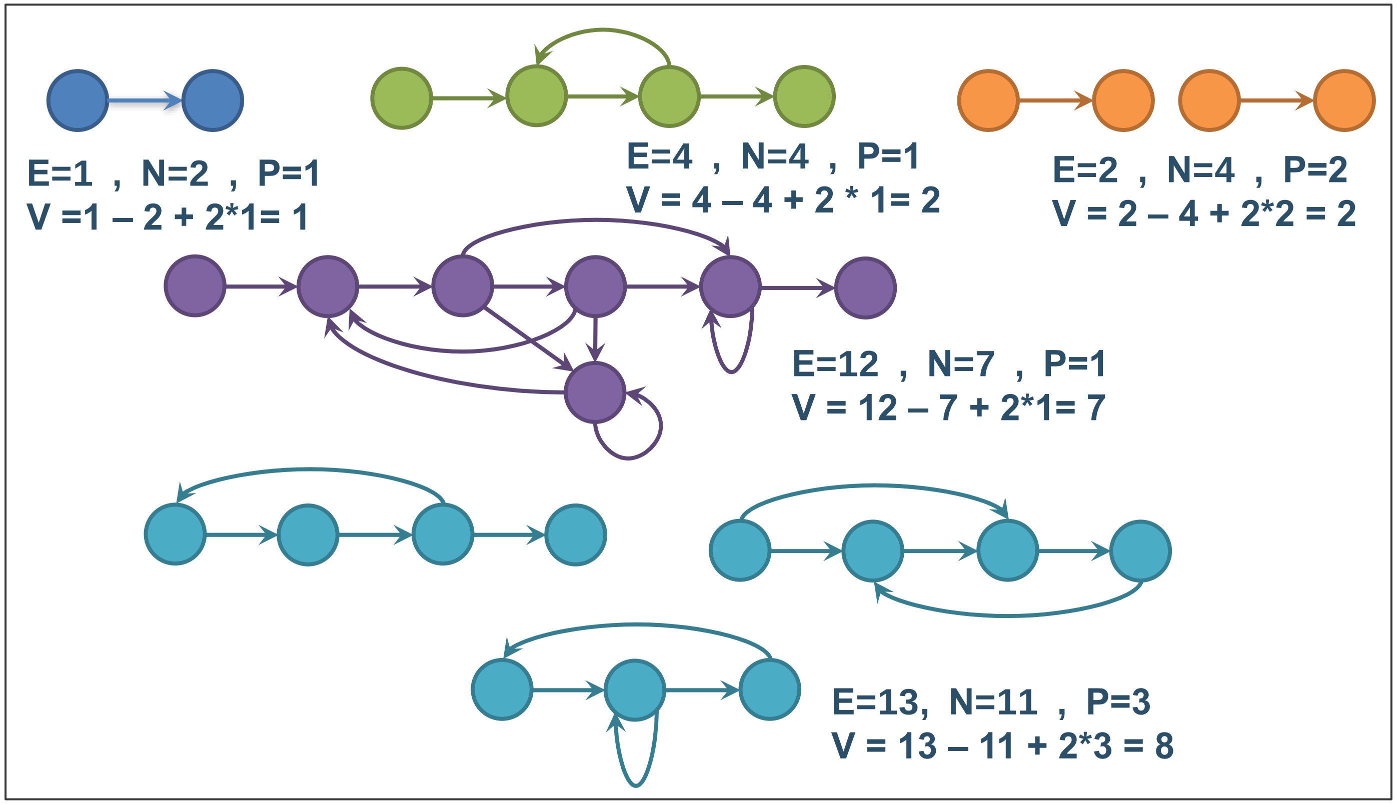
```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
          for ( j=i+1 ; j < n ; j++ ) {
                    if ( a[i] > a[j] ) {
                              t = a[i];
                              a[i] = a[j];
                              a[j] = t;
                    }
          }
   }
}
```

$V = 80 \log_2(24) \square 392$

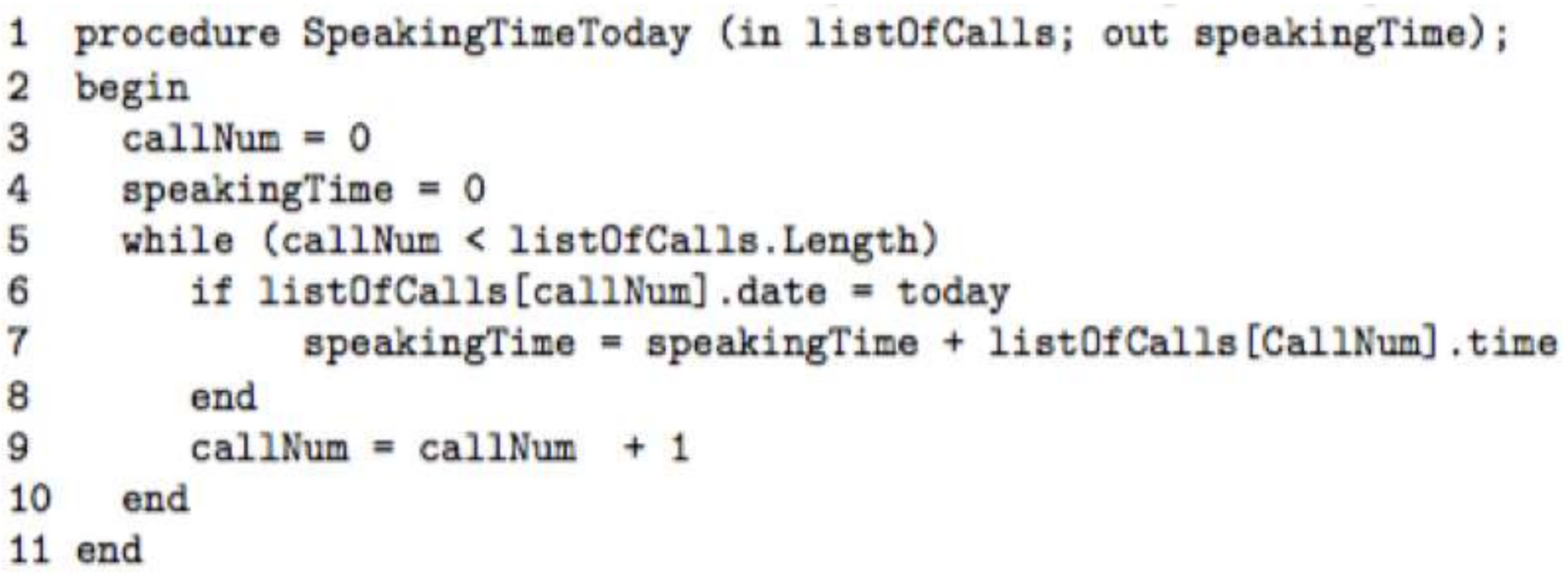
Other examples based on CFG



Other examples based on CFG



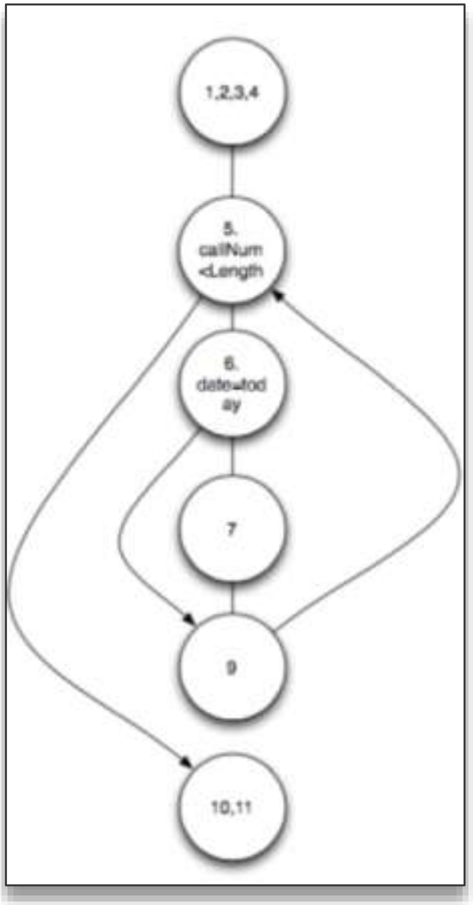
Other examples based on source code



```
procedure SpeakingTimeToday (in listOfCalls; out speakingTime);
begin
  callNum = 0
  speakingTime = 0
  while (callNum < listOfCalls.Length)
     if listOfCalls[callNum].date = today
          speakingTime = speakingTime + listOfCalls[CallNum].time
     end
     callNum = callNum  + 1
  end
end
```

Other examples based on source code

```
1  procedure SpeakingTimeToday (in listOfCalls; out speakingTime);
2  begin
3    callNum = 0
4    speakingTime = 0
5    while (callNum < listOfCalls.Length)
6       if listOfCalls[callNum].date = today
7            speakingTime = speakingTime + listOfCalls[CallNum].time
8       end
9       callNum = callNum  + 1
10   end
11 end
```

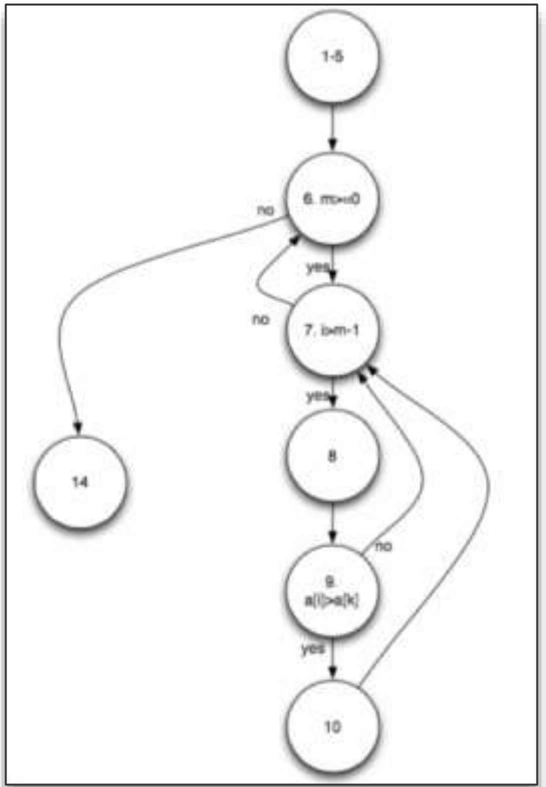


Other examples based on source code

```
public class MyBubbleSort {

 public static void bubble_srt(int array[]) {
    int n = array.length;
    int k;
    for (int m = n; m >= 0; m--) {
      for (int i = 0; i < m - 1; i++) {
        k = i + 1;
        if (array[i] > array[k]) {
           swapNumbers(i, k, array);
        }
      }
    }
  }

   private static void swapNumbers(int i, int j, int[] array) {
     int temp;
     temp = array[i];
     array[i] = array[j];
     array[j] = temp;
   }
}
```

Other examples based on source code

```
1 public class MyBubbleSort {
2
3   public static void bubble_srt(int array[]) {
4         int n = array.length;
5         int k;
6         for (int m = n; m >= 0; m--) {
7             for (int i = 0; i < m - 1; i++) {
8                 k = i + 1;
9                 if (array[i] > array[k]) {
10                     swapNumbers(i, k, array);
11                }
12            }
13        }
14   }
15
16     private static void swapNumbers(int i, int j, int[] array) {
17         int temp;
18         temp = array[i];
19         array[i] = array[j];
20         array[j] = temp;
21     }
22 }
```



McCabe Cyclomatic Number – Recommendations Khuyến cáo đối với chỉ số McCabe

- The cyclomatic number of a function should be less than 15.
  - i.e., if a function has a cyclomatic number of 15, there are at least 15 (but probably more) execution paths through it
- More than 15 paths are hard to identify and test
  - Functions containing one selection statement with many branches make up a exception
- A reasonable upper limit cyclomatic number of a file is 100

2.5. Maintainability Index

Maintainability Index (MI)
Chỉ số bảo trì được của chương trình

- Calculated with certain formulas from lines of code measures, McCabe measure and Halstead measures
- Indicates when it becomes cheaper and/or less risky to rewrite the code instead to change it
- Two variants of Maintainability Index:
  - One that contains comments (MI) and one that does not contain comments (MIwoc)
- Actually there are three measures:
  - MIwoc: Maintainability Index without comments
  - MIcw: Maintainability Index comment weight
  - MI: Maintainability Index = **MIwoc + MIcw**

MI's formula
Công thức tính chỉ số MI

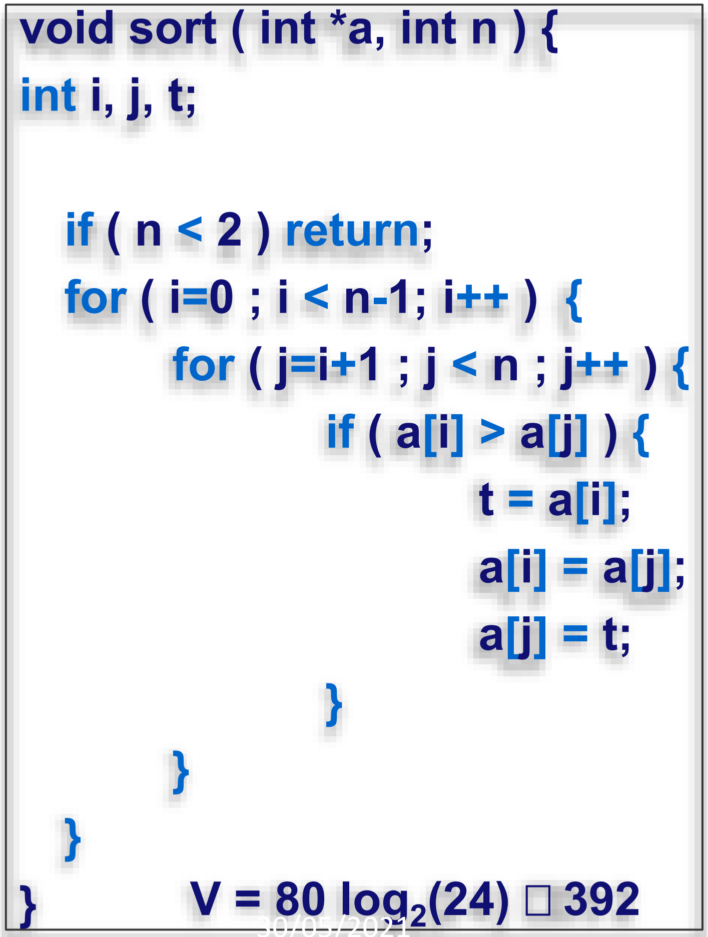
- MIwoc = 171 - 5.2 * ln(aveV) - 0.23 * aveG - 16.2 * ln(aveLOC)
  - aveV = average Halstead Volume V per module
  - aveG = average extended cyclomatic complexity v(G) per module
  - aveLOC = average count of lines LOCphy per module
  - perCM = average percent of lines of comments per module
- MIcw = 50 * sin(√2,4*perCM)

MI Recommendations
Các khuyến cáo đối với chỉ số MI

- Maintainability Index (MI, with comments) values:
  - 85 and more: good maintainability
  - 65-85: Moderate maintainability
  - < 65: Difficult to maintain with really bad pieces of code
  - big, uncommented, unstructured module can lead to a MI value can be event negative

Maintainability Index Example

- Halstead's V = 392
- McCabe's v(G) = 5
- LOC = 14
- MI = 96 --> easy to maintain!



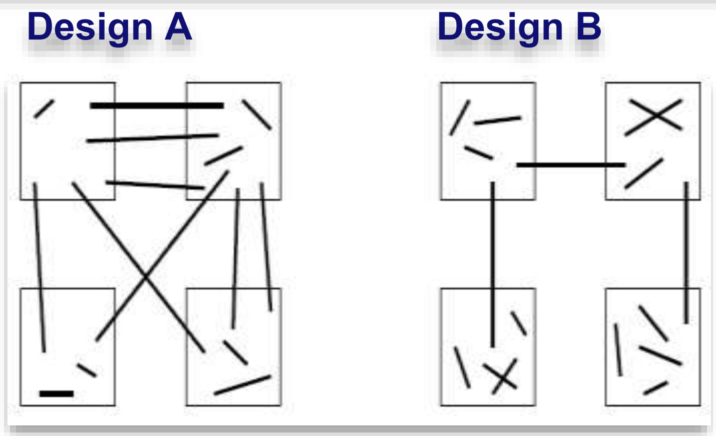
```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
         for ( j=i+1 ; j < n ; j++ ) {
                  if ( a[i] > a[j] ) {
                          t = a[i];
                          a[i] = a[j];
                          a[j] = t;
                  }
         }
   }
}
```

$V = 80 \log_2(24)$ □ 392

2.6. OOP Software Metrics

What about modularity?
Thế nào là tính mô đun hoá của chương trình

- Generally, modularity is the degree to which a system may be separated and recombined
- High cohesion and Low coupling is a good design



- **Cohesion: calls inside the module**
- **Coupling: calls between the modules**

	A	B
Cohesion	**Lo**	**Hi**
Coupling	**Hi**	**Lo**

- **Squares are modules, lines are calls, ends of the lines are functions.**
- **Which design is better?**

Modularity metrics: Fan-in and Fan-out Chỉ số về tính mô đun hoá

- Fan-in of M: number of modules calling functions in M
- Fan-out of M: number of modules called by M
- Modules with fan-in = 0
  - deadcode
  - outside of the system boundaries
- Henry and Kafura's information flow complexity
- Fan-in and fan-out can be defined for procedures and functions
  - HK: take global data structures into account:
    - fan-in: flow of information into procedures
    - fan-out: flow of information out of procedures
  - HK = SLOC x (fan-in x fan-out)^2
    - SLOC: the length (in lines of code) of the procedure without comments and blank lines

Object-oriented metrics

- Object-oriented metrics measure the structure or behaviour of an object-oriented system
- The system is viewed as a collection of objects rather than as functions/procedures with messages passed from object to object
- Some traditional metrics can be adapted to work within the OO-domain:
  - LOC, percentage of comments, Cyclomatic complexity
  - These metrics are applied within methods

Chidamber and Kemerer's metrics Độ đo của Chidamber và Kemerer

- Weighted methods per class (WMC)
- Response for a class (RFC)
- Lack of cohesion of methods (LCOM)
- Coupling between objects (CBO)
- Depth of inheritance tree (DIT)
- Number of children (NOC)

Weighted Methods per Class (WMC) Phương thức trên một lớp

- WMC is a count of the methods implemented within a class or the sum of the complexities of the methods
    - method complexity is measured by cyclomatic complexity
- The number of methods and their complexity is a predictor of how much time and effort is required to develop and maintain the class
- The larger the number of methods in class, the greater the potential impact on children
    - children inherit all of the methods defined in the parent class
- Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse

Response for a Class (RFC)
Phản hồi của một lớp

- The RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class
  - This includes all methods accessible within the class hierarchy
- This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes
- The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class
- If a large number of methods can be invokes in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester

Lack of cohesion (LCOM) (Sự thiếu độ kết dính)

- Lack of cohesion measures the dissimilarity of methods in a class by instance variables or attributes
- A highly cohesive module should stand alone
  - High cohesion indicates
    - a good class subdivision
    - simplicity and high reusability
- LCOM(C)=P-Q if P > Q and = 0 if otherwise
  - P: #pairs of distinct methods in C that do not share variables
  - Q: #pairs of distinct methods in C that share variables
- Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process
- Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion

Coupling between object classes (CBO) Sự kết nối giữa các đối tượng của các lớp

- CBO is a count of the number of other classes to which a class is coupled
  - measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends
- Excessive coupling is detrimental to modular design and prevents reuse
  - the more independent class is, the easier it is reused in another application
- The larger the number of couples, the higher the sensitivity to changes in other parts of the design
  - therefore maintenance is more difficult
- Strong coupling complicates a system since a class is harder to understand, change or correct
- Complexity can be reduced by designing systems with the weakest possible coupling between classes
  - This improves modularity and promotes encapsulation

Depth of Inheritance Tree (DIT)
Độ sâu cây kế thừa

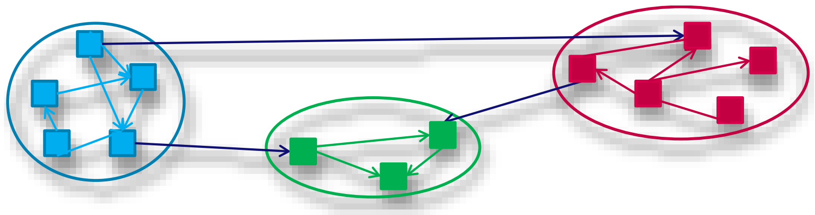
- The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes
- The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit making it more complex to predict its behavior
- Deeper trees constitute greater design complexity
  - More methods and classes are involved
  - But the greater the potential for reuse of inherited methods

Number of Children (NOC)
Số lượng lớp con

- The number of children is the number of immediate subclasses subordinate to a class in the hierarchy
- If a class has a large number of children, it may require more testing of the methods of that class, thus increasing the testing time
- NOC is an indicator of the potential influence a class can have on the system
  - The greater the NOC, the greater the likelihood of improper abstraction of the parent
  - But the greater the NOC, the greater the reuse since inheritance is a form of reuse

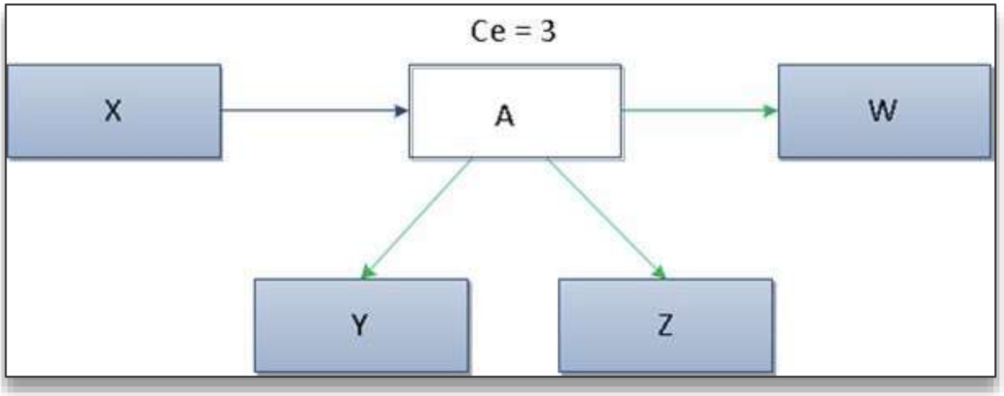
Package-level Metrics
Các độ đo ở mức gói

- Proposed by Robert Cecil Martin (Uncle Bob)
- Focus on the relationship between packages in the project
- Martin's metrics include
  - Efferent Coupling (Ce)
  - Afferent Coupling (Ca)
  - Instability (I)
  - Abstractness (A)
  - Normalized Distance from Main Sequence (D)



Efferent Coupling (CE)

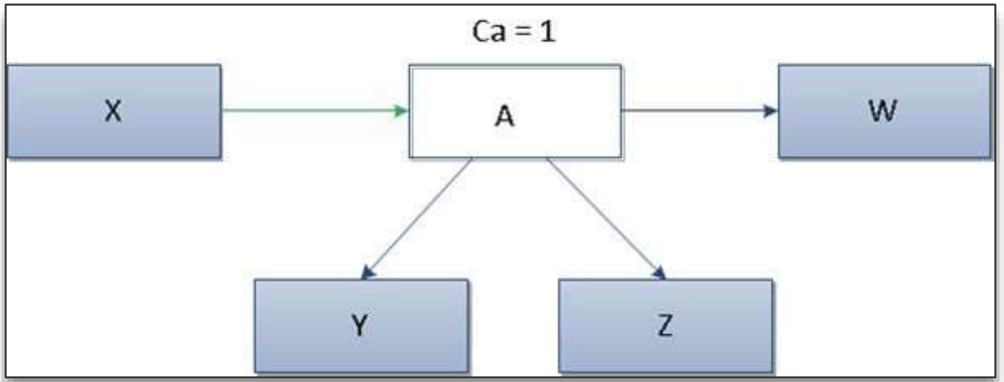
- This metric is used to measure interrelationship between classes (outgoing dependencies)
- It is a count for the number of classes in a given package which depend on the classes of other packages
- An indicator of the package's responsibility



- The high value of Ce indicates instability of a package
  - change in any of the numerous external classes can cause the need for changes to the package
- Preferred values for Ce are in the range of 0 to 20

Afferent Coupling (CA)

- Another type of dependencies between packages, i.e., incoming dependencies
- It measures the number of classes and interfaces from other packages that depend on classes in the analysed package



Instability (I)

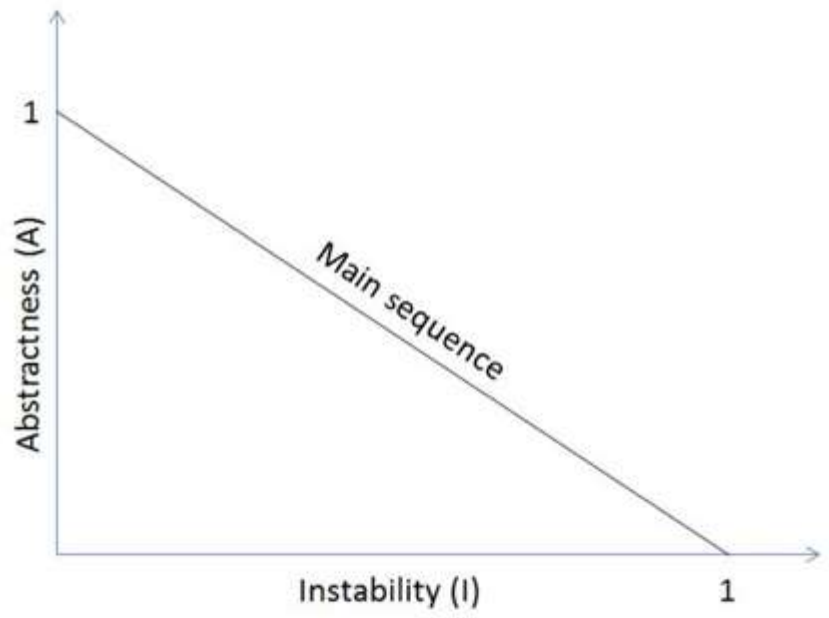
- This metric is used to measure the relative susceptibility of class to changes
- I = Ce / (Ce + Ca)
- On the basis of value of I, we can distinguish two types of components
  - The ones having many outgoing dependencies and not many of incoming ones (value of I is close to 1), which are rather unstable due to the possibility of easy changes to these packages
  - The ones having many incoming dependencies and not many of outgoing ones (value of I is close to 0), therefore they are rather more difficult in modifying due to their greater responsibility

Abstractness (A)

- This metric is used to measure the degree of abstraction of the package and is somewhat similar to the instability
- A = Ta / (Ta + Tc)
  - Ta: number of abstract classes in a package
  - Tc: number of concrete classes in this package
- Packages that are stable (metric I close to 0) should also be abstract (metric A close to 1)
- Packages that are unstable (metric I close to 1) should consists of concrete classes (metric A close to 0)

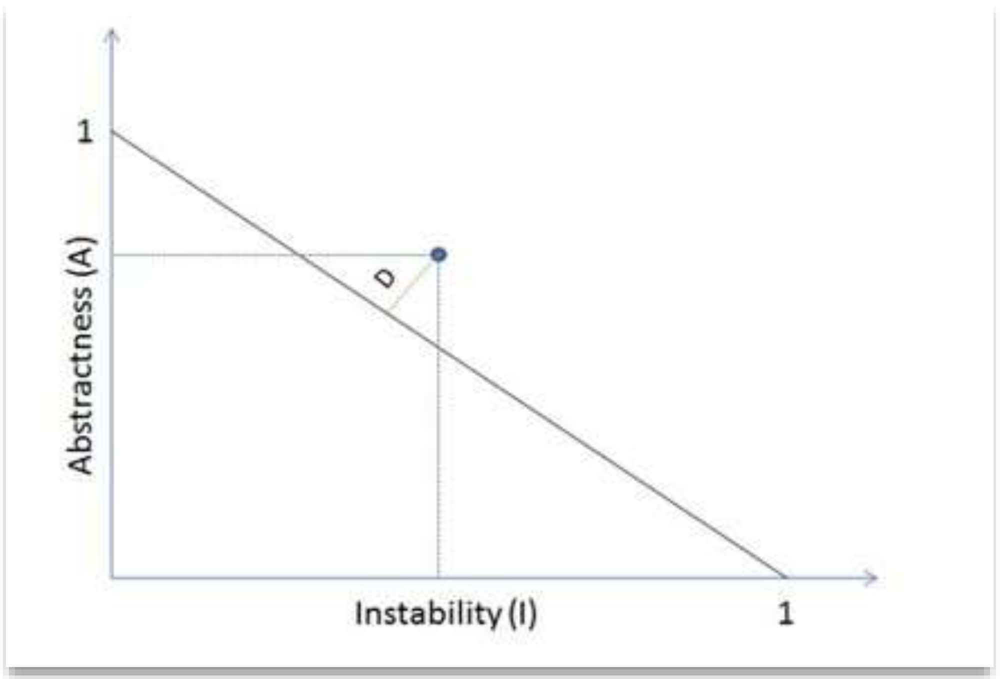
Main Sequence
Đường trình tự chính

- Combining A and I formulates the existence of main sequence
- Classes that were well designed should group themselves around the graph end points along the main sequence



Normalized distance from main sequence (D)

- This metric is used to measure the balance between stability and abstractness and is calculated using the following formula
  - D = | A + I - 1|
  - If we put a given class on a graph of the main sequence, its distance from the main sequence will be proportional to the value of D



Benefits of Normalized distance from main sequence (D) Ý nghĩa của chỉ số D

- The value of D should be as low as possible so that the components were located close to the main sequence
- A = 0 and I = 0, a package is extremely stable and concrete, the situation is undesirable because the package is very stiff and cannot be extended
- A = 1 and I = 1, rather impossible situation because a completely abstract package must have some connection to the outside, so that the instance that implements the functionality defined in abstract classes contained in this package could be created

Summaries
Tổng kết

- Students must understand the motivation of using software metrics to improve the quality of source code and also to improve the quality of software systems
- Give a general perspective about software metrics that are commonly used in software engineering
    - General metrics of lines of code
    - McCabe Cyclomatic Number
    - Halstead’s metrics
    - Maintainability Index
    - OOP metrics

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attention!!!

soict.hust.edu.vn/ fb.com/groups/soict