Chapter 7

Coding

Declaration

- These slides are made for UIT, BU students only. I am not holding any copy write of it as I had collected these study materials from different books and websites etc. I have not mentioned those to avoid complexity.

Coding

- The coding is concerned with translating design specifications into source code.
- The good programming should ensure the ease of debugging, testing and modification.
- This is achieved by making the source code as clear and straightforward as possible.
- An old saying is “Simple is great”.
- Simplicity, clarity and elegance are the hallmarks of good programs.
- Obscurity, cleverness, and complexity are indications of inadequate design.
- Source code clarity is enhanced by structured coding techniques, by good coding style, by appropriate supporting documents, by good internal comments etc.

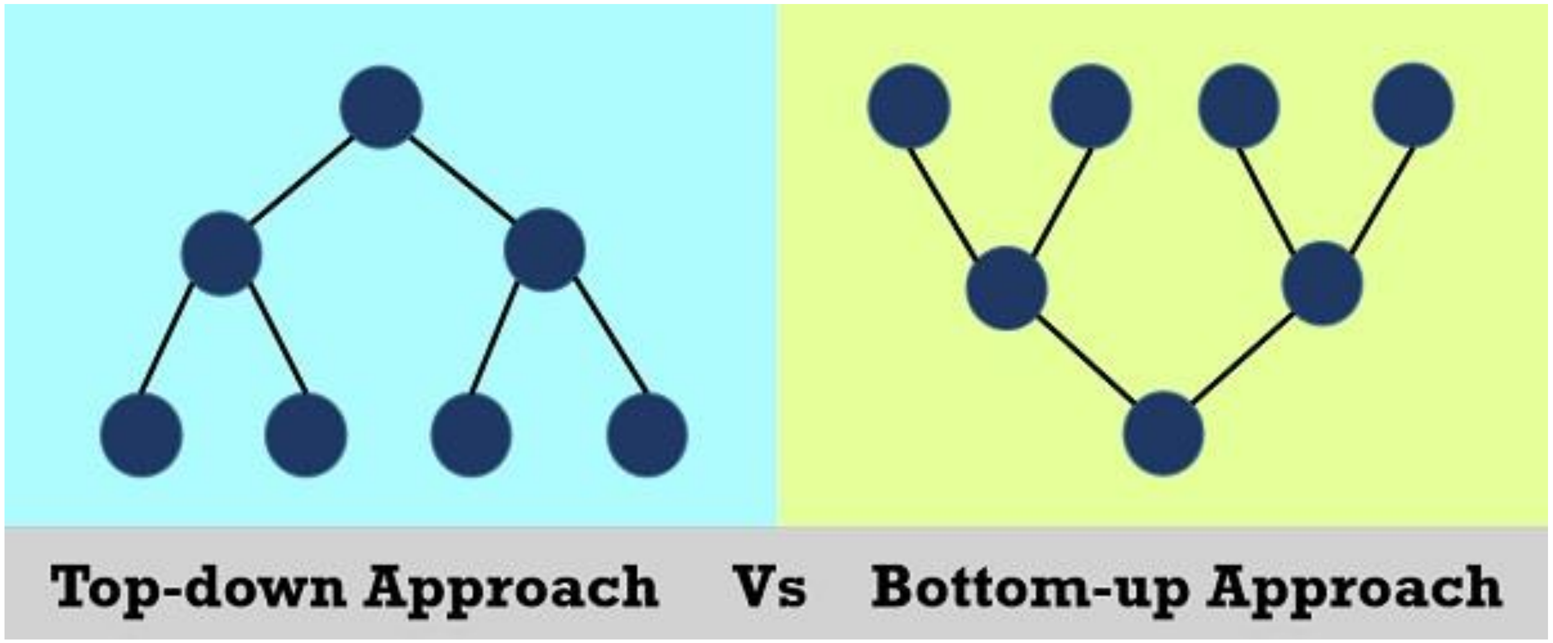
Coding

- Production of high quality software requires that the programming team should have a thorough understanding of duties and responsibilities and should be provided with a well defined set of requirements, an architectural design specification, and a detailed design description.

Top-down and Bottom up approach

- When talking in terms of computer science and programming, the algorithms we use to solve complex problems in a systematic and controlled way are designed on the basis of two approaches that is Top-down and Bottom-up approach.
- The ideology behind top-down approach is, a bigger problem is divided into some smaller sub-problems called *modules,* these modules are then solved individually and then integrated together to get the complete solution to the problem.
- In bottom-up approach on the other hand, the process starts with elementary modules and then combining together to get the desired result. Let us now quickly see in brief what these two approaches has to offer, how they differ from each other and what are the similarities.

Top-down and Bottom up approach



Top-down approach

- The basic idea in top-down approach is to break a complex algorithm or a problem into smaller segments called modules, this process is also called as *modularization.*
- The modules are further decomposed until there is no space left for breaking the modules without hampering the originality. The uniqueness of the problem must be retained and preserved. The decomposition of the modules is restricted after achieving a certain level of modularity.
- The top-down way of solving a program is step-by-step process of breaking down the problem into chunks for organizing and solving the sole problem. The C - programming language uses the top-down approach of solving a problem in which the flow of control is in the downward direction.

Bottom up approach

- As the name suggests, this method of solving a problem works exactly opposite of how the top-down approach works. In this approach we start working from the most basic level of problem solving and moving up in conjugation of several parts of the solution to achieve required results. The most fundamental units, modules and sub-modules are designed and solved individually, these units are then integrated together to get a more concrete base to problem solving.
- This bottom-up approach works in different phases or layers. Each module designed is tested at fundamental level that means unit testing is done before the integration of the individual modules to get solution.

Difference between Top-down and Bottom-up Approach

Top-Down Approach	Bottom-Up Approach
Divides a problem into smaller units and then solve it.	Starts from solving small modules and adding them up together.
This approach contains redundant information.	Redundancy can easily be eliminated.
A well-established communication is not required.	Communication among steps is mandatory.
The individual modules are thoroughly analysed.	Works on the concept of data-hiding and encapsulation.
Structured programming languages such as C uses top-down approach.	OOP languages like C++ and Java, etc. uses bottom-up mechanism.

Difference between Top-down and Bottom-up Approach

Top-Down Approach	Bottom-Up Approach
Relation among modules is not always required.	The modules must be related for better communication and work flow.
Primarily used in code implementation, test case generation, debugging and module documentation.	Finds use primarily in testing.

Structured Programming

- In the process of coding, the lines of code keep multiplying, thus, size of the software increases. Gradually, it becomes next to impossible to remember the flow of program. If one forgets how software and its underlying programs, files, procedures are constructed, it then becomes very difficult to share, debug, and modify the program. The solution to this is structured programming. It encourages the developer to use subroutines and loops instead of using simple jumps in the code, thereby bringing clarity in the code and improving its efficiency Structured programming also helps programmer to reduce coding time and organize code properly.

Structured Programming

- Structured programming states how the program shall be coded. It uses three main concepts:

1. **Top-down analysis** - A software is always made to perform some rational work. This rational work is known as problem in the software parlance. Thus it is very important that we understand how to solve the problem. Under top-down analysis, the problem is broken down into small pieces where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.

Structured Programming

2. **Modular Programming** - While programming, the code is broken down into smaller group of instructions. These groups are known as modules, subprograms, or subroutines. Modular programming based on the understanding of top-down analysis. It discourages jumps using ‘goto’ statements in the program, which often makes the program flow non-traceable. Jumps are prohibited and modular format is encouraged in structured programming.

3. **Structured Coding** - In reference with top-down analysis, structured coding sub-divides the modules into further smaller units of code in the order of their execution. Structured programming uses control structure, which controls the flow of the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

Coding Documentation

- **Code documentation** is a manual-cum-guide that helps in understanding and correctly utilizing the software code. The coding standards and naming conventions written in a commonly spoken language in code documentation provide enhanced clarity for the designer. Moreover, they act as a guide for the software maintenance team (this team focuses on maintaining software by improving and enhancing the software after it has been delivered to the end user) while the software maintenance process is carried out. In this way, code documentation facilitates code reusability.

Coding Documentation

- While writing a software code, the developer needs proper documentation for reference purposes. Programming is an ongoing process and requires modifications from time to time. When a number of software developers are writing the code for the same software, complexity increases. With the help of documentation, software developers can reduce the complexity by referencing the code documentation. Some of the documenting techniques are comments, visual appearances of codes, and programming tools. **Comments** are used to make the reader understand the logic of a particular code segment. The **visual appearance of a code** is the way in which the program should be formatted to increase readability. The **programming tools** in code documentation are algorithms, flowcharts, and pseudo-codes.

Coding Documentation

- Code documentation contains source code, which is useful for the software developers in writing the software code. The code documents can be created with the help of various coding tools that are used to auto-generate the code documents. In other words, these documents extract comments from the source code and create a reference manual in the form of text or HTML file. The auto-generated code helps the software developers to extract the source code from the comments. This documentation also contains application programming interfaces, data structures, and algorithms. There are two kinds of code documentation, namely, internal documentation and external documentation.

Coding Documentation

- Documentation which focuses on the information that is used to determine the software code is known as internal documentation. It describes the data structures, algorithms, and control flow in the programs. There are various guidelines for making the documentation easily understandable to the reader.

Coding Documentation

- Generally, internal documentation comprises the following information.
  - Name, type, and purpose of each variable and data structure used in the code
  - Brief description of algorithms, logic, and error-handling techniques
  - Information about the required input and expected output of the program
  - Assistance on how to test the software
  - Information on the up-gradations and enhancements in the program.

Coding Documentation

- Documentation which focuses on general description of the software code and is not concerned with its detail is known as external documentation. It includes information such as function of code, name of the software developer who has written the code, algorithms used in the software code, dependency of code on programs and libraries, and format of the output produced by the software code. Generally, external documentation includes structure charts for providing an outline of the program and describing the design of the program.

Verification and Validation

- **Verification** is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- **Validation** is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .
- **Testing= Verification+Validation**

Software Metrics

Software Metrics: What and Why ?

1. How to measure the size of a software?
2. How much will it cost to develop a software?
3. How many bugs can we expect?
4. When can we stop testing?
5. When can we release the software?
6. What is the complexity of a module?
7. What is the module strength and coupling?
8. What is the reliability at the time of release?
9. Which test technique is more effective?
10. Are we testing hard or are we testing smart?
11. Do we have a strong program or a week test suite?

Software Metrics

- Pressman explained as “A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of the product or process”.
- Measurement is the act of determine a measure.
- The metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- Fenton defined measurement as “ it is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules”.

Software Metrics

- Software metrics can be defined as "*The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products*".

Software Metrics: Areas of Applications

- The most established area of software metrics is cost and size estimation techniques.
- The prediction of quality levels for software, often in terms of reliability, is another area where software metrics have an important role to play.
- The use of software metrics to provide quantitative checks on software design is also a well established area.

Software Metrics : Categories

1. **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.
2. **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:
   - effort required in the process
   - time to produce the product
   - effectiveness of defect removal during development
   - number of defects found during testing
   - maturity of the process
3. **Project metrics:** describe the project characteristics and execution. Examples are :
   - number of software developers
   - staffing pattern over the life cycle of the software
   - cost and schedule
   - productivity

Token Count

The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2$$

where

η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

Token Count

The length of the program in the terms of the total number of tokens used is

$$N = N_1 + N_2$$

where

N : program length

N_1 : total occurrences of operators

N_2 : total occurrences of operands

Token Count

Volume

$$V = N * \log_2 \eta$$

The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used.

Program Level (Halstead's software metrics)

$$L = V^* / V$$

The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size).

V = actual volume of the code
V* = **potential minimum volume** of the program if it were written in the most optimal way

Token Count

Program Difficulty

$$D = 1 / L$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

Effort

$$E = V / L = D * V$$

The unit of measurement of E is elementary mental discriminations.

The term "elementary mental discrimination" as a **unit of measurement for E** (often denoted as **Effort** in Halstead’s software metrics) represents the amount of mental effort required to understand or implement a particular piece of software.

Token Count

- Estimated Program Length

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

- Potential Volume

$$V^* = (2+\eta_2^*)\log_2(2+\eta_2^*)$$ Here η_2^* = Unique input and output

- Estimated Program Level / Difficulty

Halstead offered an alternate formula that estimate the program level.

$$\hat{L} = 2\eta_2 / (\eta_1 N_2)$$

where

$$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2}$$

Token Count

- Effort and Time

$$E = V / \hat{L} = V * \hat{D}$$

$$= (n_1 N_2 N \log_2 \eta) / 2\eta_2$$

$$T = E / \beta$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing.

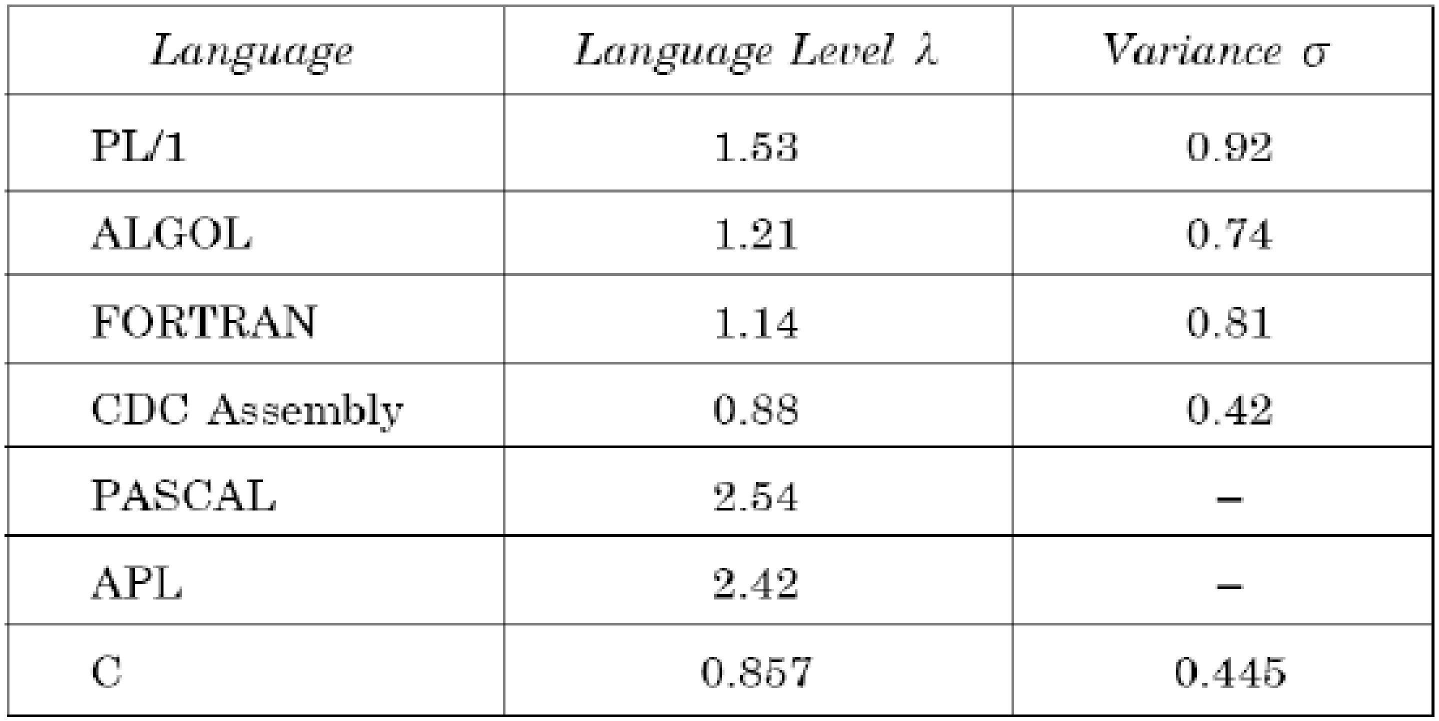
Token Count

- Language Level

$$\lambda = L \times V^* = L^2 V$$

Using this formula, Halstead and other researchers determined the language level for various languages as shown in Table 1.

Token Count



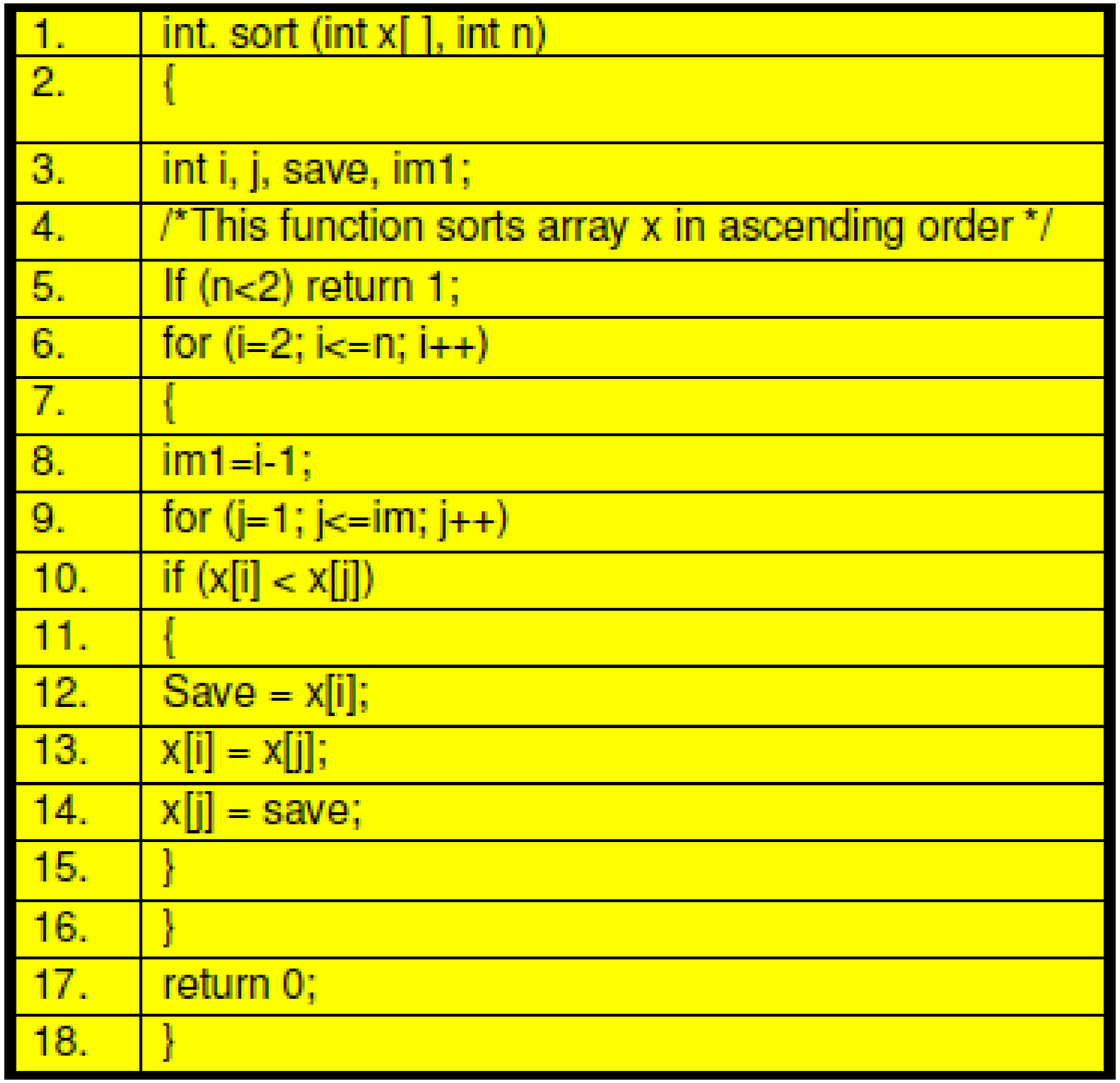
Language	*Language Level* λ	*Variance* σ
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	–
APL	2.42	–
C	0.857	0.445

Table 1: Language levels

Example

- Consider the sorting program in the next figure . List out the operators and operands and also calculate the values of software science measures like Software Metricsη, N, V, E, λ *etc*.

Example



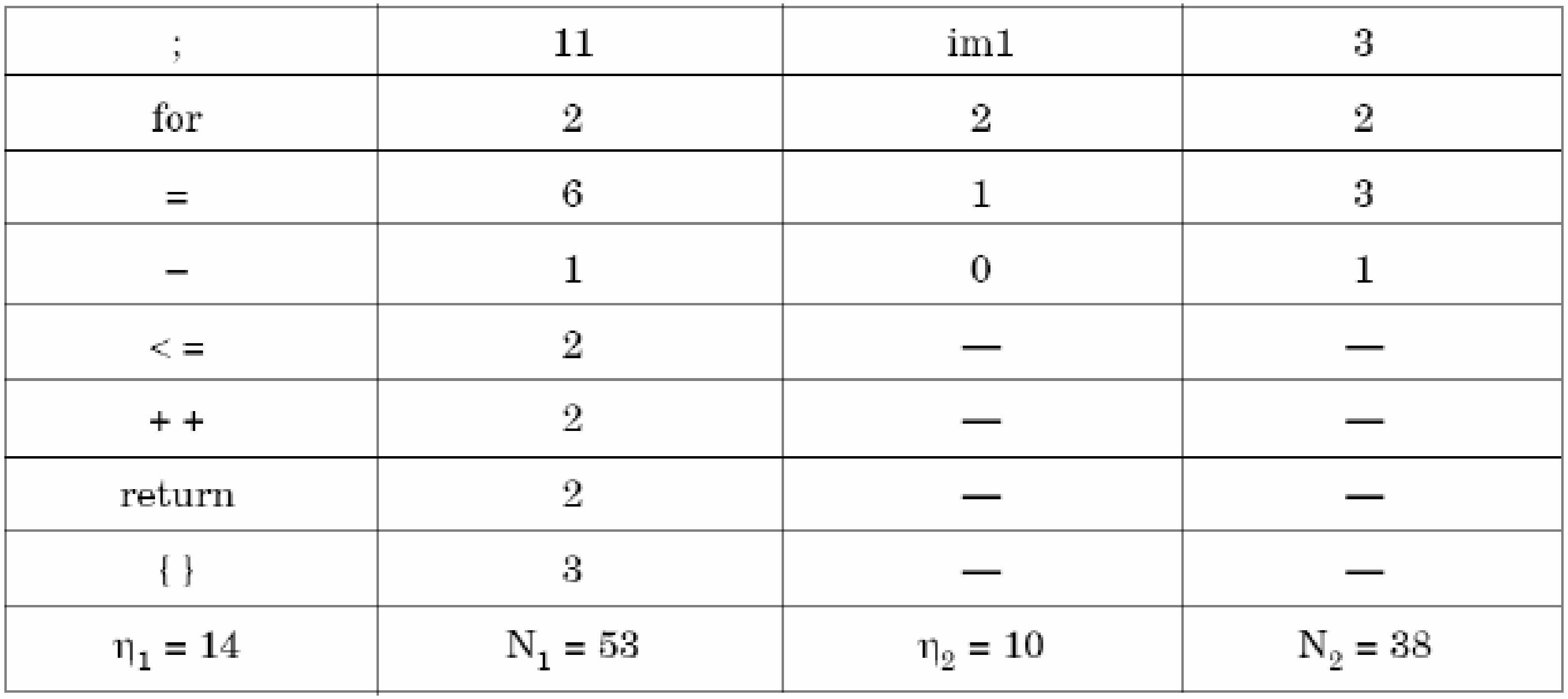
1.	int. sort (int x[ ], int n)
2.	{
3.	int i, j, save, im1;
4.	/*This function sorts array x in ascending order */
5.	If (n<2) return 1;
6.	for (i=2; i<=n; i++)
7.	{
8.	im1=i-1;
9.	for (j=1; j<=im; j++)
10.	if (x[i] < x[j])
11.	{
12.	Save = x[i];
13.	x[i] = x[j];
14.	x[j] = save;
15.	}
16.	}
17.	return 0;
18.	}

Example

The list of operators and operands

Operators	*Occurrences*	*Operands*	*Occurrences*
int	4	SORT	1
( )	5	x	7
,	4	n	3
[ ]	7	i	8
if	2	j	7
$<$	2	save	3

Example



;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
< =	2	—	—
+ +	2	—	—
return	2	—	—
{ }	3	—	—
$\eta_1 = 14$	$N_1 = 53$	$\eta_2 = 10$	$N_2 = 38$

Example

Here N_1=53 and N_2=38. The program length $N=N_1+N_2=91$

Vocabulary of the program $\eta = \eta_1 + \eta_2 = 14 + 10 = 24$

Volume $V = N \times \log_2 \eta$

$= 91 \times \log_2 24 = 417$ bits

The estimated program length $\hat{N}$ of the program

$= 14 \log_2 14 + 10 \log_2 10$

$= 14 * 3.81 + 10 * 3.32$

$= 53.34 + 33.2 = 86.45$

Example

$\eta_2^* = 3$ {x: array holding the integer to be sorted. This is used both as input and output}.

{N: the size of the array to be sorted}.

The potential volume $V^* = 5 \log_2 5 = 11.6$

Since Program Level $L = V^* / V$

Example

$$= \frac{11.6}{417} = 0.027$$

Program Difficulty D = I / L

$$= \frac{1}{0.027} = 37.03$$

Estimated program level

$$\hat{L} = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2} = \frac{2}{14} \times \frac{10}{38} = 0.038$$

Example

We may use another formula

$$\hat{V} = V \times \hat{L} = 417 \times 0.038 = 15.67$$

$$\hat{E} = V / \hat{L} = \hat{D} \times V$$

$$= 417 / 0.038 = 10973.68$$

Therefore, 10974 elementary mental discrimination are required to construct the program.

$$T = E / \beta = \frac{10974}{18} = 610 \text{seconds} = 10 \text{minutes}$$

This is probably a reasonable time to produce the program, which is very simple

The Sharing of Data Among Modules

A program normally contains several modules and share coupling among modules. However, it may be desirable to know the amount of data being shared among the modules.

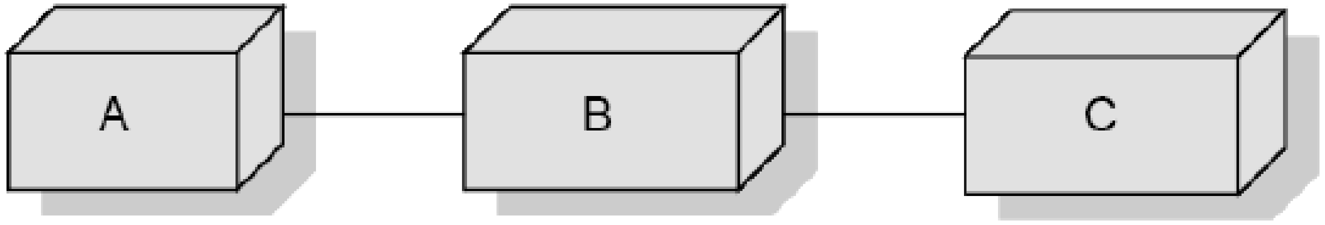


Fig.10: Three modules from an imaginary program

The Sharing of Data Among Modules

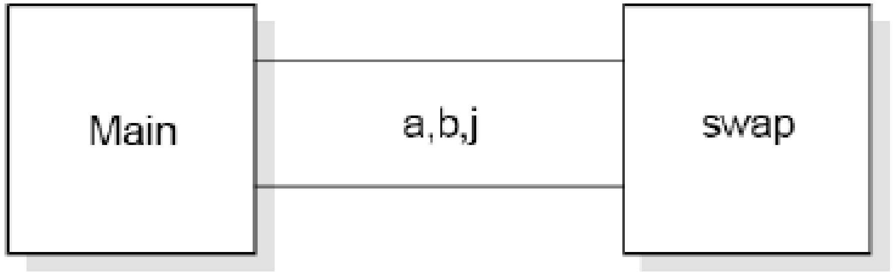


Fig.12: The data shared in program bubble

Information Flow Metrics

Component	: Any element identified by decomposing a (software) system into its constituent parts.
Cohesion	: The degree to which a component performs a single function.
Coupling	: The term used to describe the degree of linkage between one component to others in the same system.

The Basic Information Flow Model

Information Flow metrics are applied to the Components of a system design. Fig. 13 shows a fragment of such a design, and for component 'A' we can define three measures, but remember that these are the simplest models of IF.

1. 'FAN IN' is simply a count of the number of other Components that can call, or pass control, to Component A.
2. 'FANOUT' is the number of Components that are called by Component A.
3. This is derived from the first two by using the following formula. We will call this measure the INFORMATION FLOW index of Component A, abbreviated as IF(A).

$$IF(A) = [\text{FAN IN}(A) \times \text{FAN OUT}(A)]^2$$

The Basic Information Flow Model

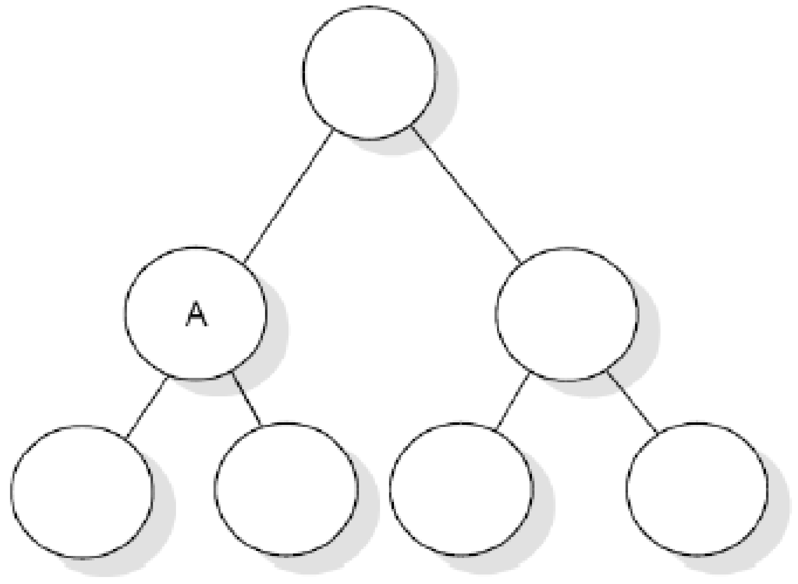


Fig.13: Aspects of complexity

The Basic Information Flow Model

The following is a step-by-step guide to deriving these most simple of IF metrics.

1. Note the level of each Component in the system design.
2. For each Component, count the number of calls so that Component – this is the FAN IN of that Component. Some organizations allow more than one Component at the highest level in the design, so for Components at the highest level which should have a FAN IN of zero, assign a FAN IN of one. Also note that a simple model of FAN IN can penalize reused Components.
3. For each Component, count the number of calls from the Component. For Component that call no other, assign a FAN OUT value of one.

cont

The Basic Information Flow Model

4. Calculate the IF value for each Component using the above formula.

5. Sum the IF value for all Components within each level which is called as the LEVEL SUM.

6. Sum the IF values for the total system design which is called the SYSTEM SUM.

7. For each level, rank the Component in that level according to FAN IN, FAN OUT and IF values. Three histograms or line plots should be prepared for each level.

8. Plot the LEVEL SUM values for each level using a histogram or line plot.

This image cannot currently be displayed.

End of Chapter 7

Questions?