Software Engineering : UNIT-3

Modularity :- A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

Modularity is a clearly a desirable property in a system. Modularity helps in a system debugging - isolating the system problem to a component is easier if the system is modular - in system repair - changing a part of the system is easy as it affects few other parts - and in system building - a modular system can be easily built by "putting its module together"

Modularity deals in partition the whole into pieces.

Examples :- A pen can be broken into pieces.

→ Modularity is property which allows to decompose system into cohesive and loosely coupled modules.

Summary regarding Modularity & Hierarchy

→ Object models helps to harness power of object oriented programming languages.
→ Reusability
→ Stability
→ Reduce risk
→ Confidence

Scanned by CamScanner

Design Notation and Specification

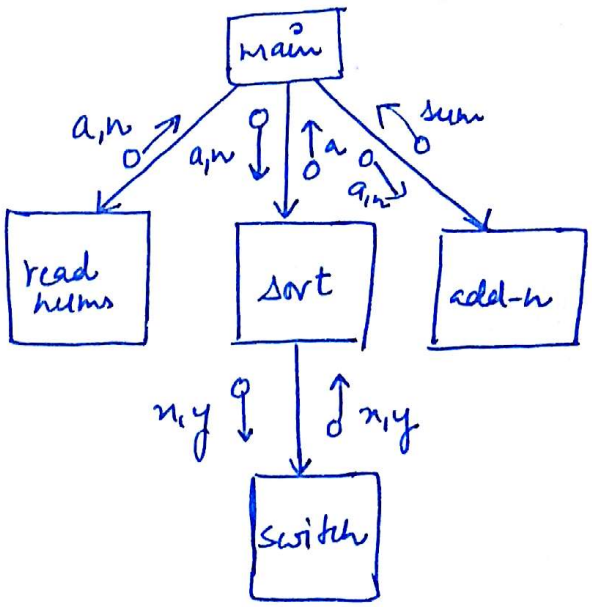
Structure Charts

→ Can be used to represent a function-oriented design.
→ It is made up of the modules of that program together with the interconnections between modules.
→ The structure chart of a program is a graphical representation of its structure.
→ In a structure chart a module is represented by a box with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the subordinate of A and A is called the superordinate of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail)

Example :-

```
main()
{ int sum,n, N, a[MAX];
  readnums(a,&N); sort(a,N); scanf(&N);
  sum = add-n(a,n); printf(sum);
}
readnums(a,N)
int a[], *N;
{
}
sort(a,N)
int a[], N;
{ if(a[i]>a[t]) switch(a[i],a[t]);
}
/* Add the first n numbers of a */
add-n(a,n)
int a[],n;
{ :
}
```

Structure Chart of the sort program

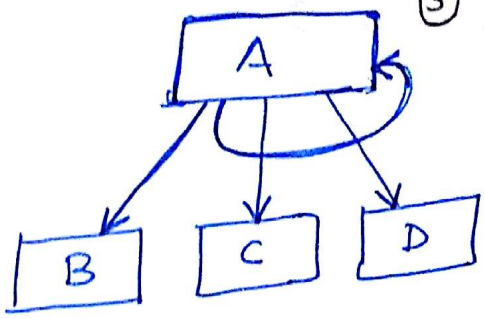


The structure chart of the sort program

Scanned by CamScanner

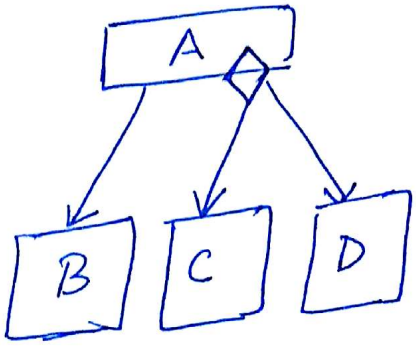
Example:-

Let us consider a situation module A has subordinates B, C, and D and A repeatedly calls the module C and D. This can be represented by a looping arrow around the arrows joining the subordinates C and D to A. All the subordinate modules activated within a common loop are enclosed in the same looping arrow.



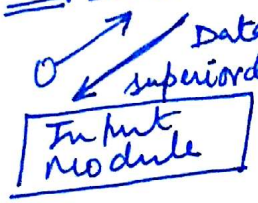
Example:-

If the invocation of modules C and D in module A depends on the outcome of some decision, i.e. represented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond

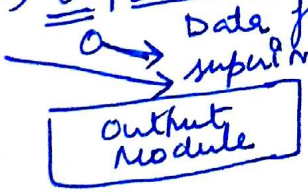


Types of Module :-

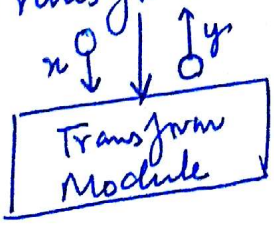
(a) Input Module :- Module that obtain information from their sub-ordinates and then pass it to their superordinate. This type of module is called Input module.



(b) Output Module :- ~~Out~~ Module that take information from their superordinate and pass it on to its subordinates.

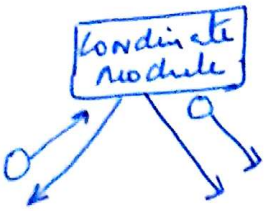


(c) Transform Module :- Module that exist solely for the sake of transforming data into some other form. Such a module is called a transform module.

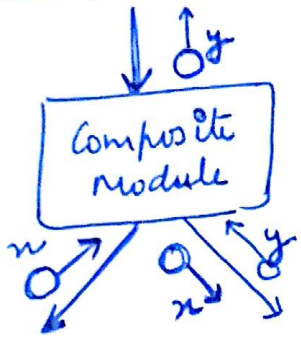


Scanned by CamScanner

(d) Co-ordinate Modules :- Modules that flow that data to and from different sub-ordinates. Such modules are called co-ordinate modules.



(e) Composite Modules :- Module is an input module from the point of view of its superordinate, as it feeds the data Y to the superordinate. Internally, A is a co-ordinate module and views its job as getting data X from one sub-ordinate and passing it to another sub-ordinate, which converts it to Y.



Ans: Draw the structure chart for the following program:

```
main()
{ int x,y;
  x=0; y=0;
  a(); b(); }

a()
{ x= x+y; y=y+5; }
b()
{ x=x+5; y=y+x; a(); }
```

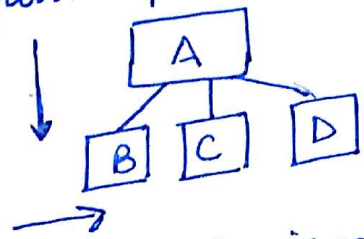
How would you modify this program to improve the modularity?

Scanned by CamScanner

Scanned by CamScanner

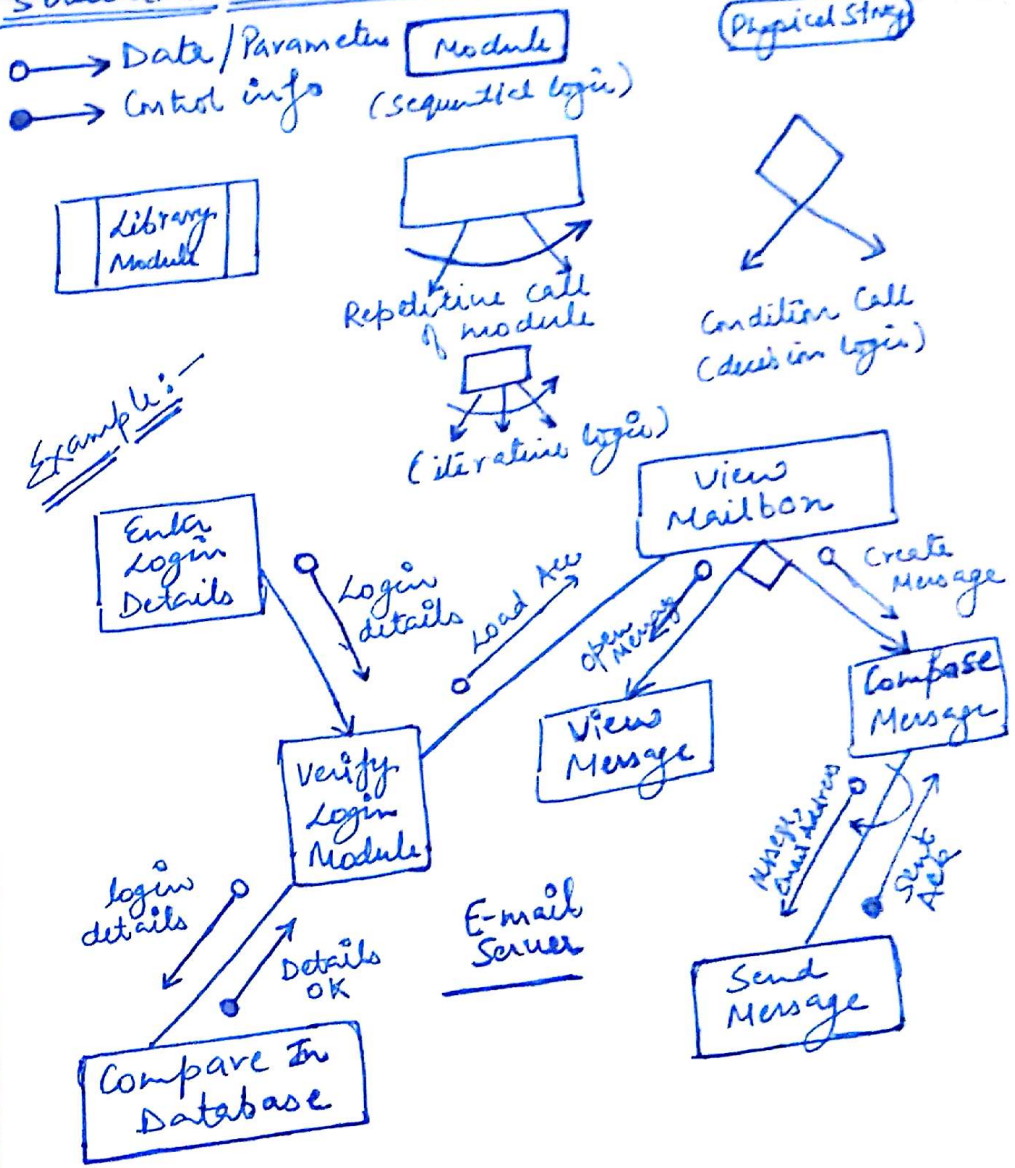
Function Oriented Design

→ Hierarchical Representation of System
→ Partitions the systems into Black Boxes
  Functionality is known to user but inner details are not known.
  • Inputs are given to Black Boxes and appropriate outputs generated
→ using Black Boxes reduces complexity of design
→ Modules at top level call modules at lower level.
→ Components are read from Top to Bottom and Left to Right



→ when a module calls another, it views the called module as black box, passing reqd. parameters and receiving results.

Structure Chart Notations



Steps in drawing a structure chart

(a) Review the DFD's and Object modules
(b) Identify modules and relationships
(c) Add couples, loops, and conditions
(d) Analyze the structure chart, the DFD's, and the data dictionary.

Summary — Structure Charts

(a) Shows the relationships among program modules
(b) Structure Chart consists of:
  (a) Modules – Related program code organized into small units that are easy to understand and maintain
  (b) Data couples – Data passed between modules
  (c) Control couples – Data passed between modules that indicates a condition or action to another module (e.g. End of File)
  (d) Conditions – determines what subordinate module a control module will run
  (e) Loops – Indicates one or more sub-ordinate modules repeated.

Scanned by CamScanner

Function Oriented Design: Pseudo Code *

- useful in both preliminary and detailed design phases.
- system description using short English like phrases describing the function
- Keywords and Id Indentation :- describes the flow of control
- English phrases & processing actions performed
- Advantage
  - used as a replacement for flow chart
  - decrease the amount of documentation required

Example 1: Program to print sum of two numbers.

```
Void sum (int a, int b)
{ int c;
  c=a+b;
  Cout << c;
}
```

```
Begin:
Procedure: Sum
Pre condition: integer a & b
declare variable c
design sum of a and b to c
Print c
End.
```

Example 2: Program to print fibonacci up to n numbers.

```
Void function fibonacci
Get value of n;
Set value of a to 1:
Set value of b to 1:
Initialize i to 0
for ( i=0; i<n; i++)
  { if a is greater than b
    { Increase b by a;
      Print b;
    }
    else if b greater than a
    { Increase a by b;
      print a;
    }
  }
```

* Pseudo code in context of Software Engineering

Scanned by CamScanner

Module Level Concepts :-

→ A module is a logically separable part of a program. It is a program unit that is discrete and identifiable with respect to compiling and loading.

→ To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately. In a system using functional abstraction, coupling and cohesion are two modularization criteria, which are often used together.

Coupling :-

→ Coupling between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections.

→ Coupling is an abstract concept and is not easily quantifiable. So, no formulas can be given to determine the coupling between two modules.

→ Major factors influencing coupling between modules.
(a) Types of connections between modules
(b) The complexity of the interface
(c) The type of information flow between modules.

→ Coupling increases with the complexity and obscurity of the interface between modules. To keep coupling low we would like to minimize the no. of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules.

→ Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example :- Complexity of the entire interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items. Essentially, we should keep the interface of a module as simple and small as possible.

Scanned by CamScanner

→ The type of information flow along the interfaces is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction.

Transfer of data information means that a module passes as input some data to another module and gets in return some data output. This allows a module to be treated as a simple input - the output data.

→ Interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data.

→ Coupling is considered highest if the data is hybrid, i.e. some data items and some control items are passed between modules.

Effect of these three factors on coupling is summarized in Table given below:-

	Interface Complexity	Type of connection	Type of communication
Low	Simple obvious	To module by name	Data Control
High	Complicated obscure	To internal elements	Hybrid

Factors affecting Coupling

What is Coupling?

→ Coupling is the measure of degree of interdependence between modules

→ High Coupling : strongly inter-related / dependent modules
Low Coupling : Independent modules

→ Low Coupling is desired because high Coupling leads to more errors + it makes isolation (debugging) of errors very difficult.

How module become coupled / dependent ?

↳ when modules share data / exchange data or they make calls to each other.

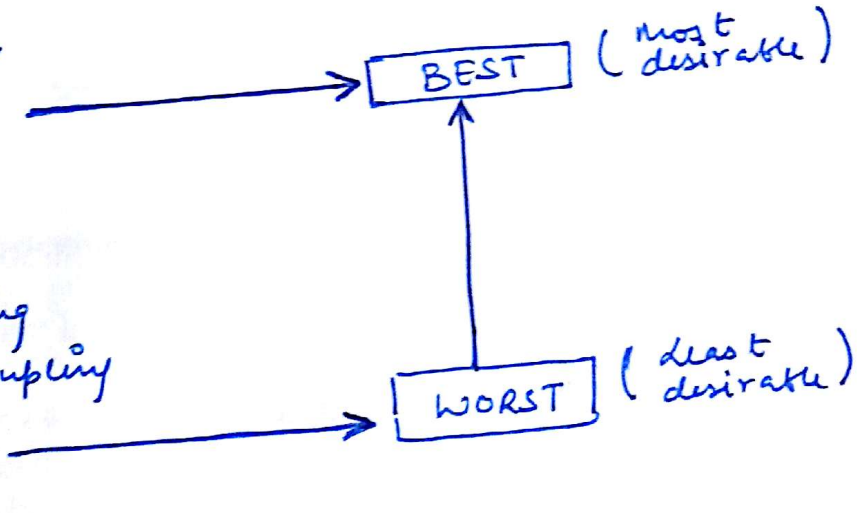
How to Control Coupling ?

↳ Control the amount of information exchanged b/w the modules

↳ Pass data not control information.

Types of Coupling —

- Data Coupling
- Stamp Coupling
- Control Coupling
- External Coupling
- Communication Coupling
- Control Coupling



1) Data Coupling :-

→ Communication b/w modules occur by only passing the necessary data (No Control info)

→ Data passed using parameters

Scanned by CamScanner

2) STAMP Coupling-
It occurs when complete data structure is passed from 1 module to another
Drawback : The entire data structure is ~~used~~ not required by receiving module, only a part of the data structure is needed

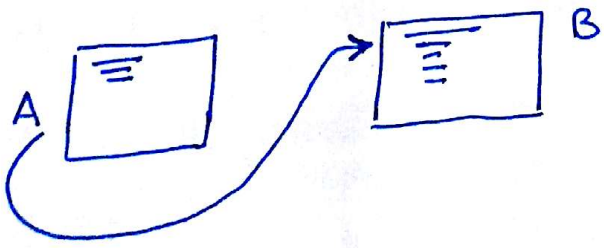
3) Control Coupling-
Communication b/w modules occurs by passing control info or a module control the flow of another module.
eg:- flags, data i.e set by 1 module is used by another module.

4) External Coupling-
Dependency of a module on another module which is present in a ~~parent~~ external to the system

5) Common Coupling-
when 2 modules have common shared data / global data (accessed by both)
→ shared data makes error isolation difficult OR evaluation of change to shared data is difficult.

6) Content Coupling-
It occurs when control is passed from one module to middle of another OR a module changes data of another module

A module modifies another module



Scanned by CamScanner

Cohesion :-

→ Cohesion of a module represents how tightly bound the internal elements of the module are to one another.

→ Greater the cohesion of each module in the system, the lower the coupling between module is.

There are several kinds of cohesion :-

(a) Co-incidental
(b) Logical
(c) Temporal
(d) Procedural
(e) Communicational
(f) Sequential
(g) Functional

⟹ Lowest level :- Co-incidental
⟹ Highest level :- Functional

Co-incidental :- It occurs when there is no meaningful relationship among the elements of a module.

Logical :- A module has logical if there is some logical relationship between the elements of a module, and the elements perform functions that fall in the same logical class. A typical example of this kind of cohesion is a module that performs all the inputs or all the ~~inputs~~ outputs.

Temporal :- Temporal cohesion is the same as logical cohesion, except that the elements are also related in time and are executed together. modules that perform activities like initialization, clean-up and termination are usually temporally bound. Temporal cohesion is higher than logical cohesion, because the elements are all executed together.

Scanned by CamScanner

Procedural :- A procedurally cohesive module contains elements that belong to a common procedural unit. For example :- a loop or a sequence of decision statements in a module may be combined to form a separate module. Procedurally cohesive modules often occur when modular structure is determined from some form of flow chart. A module with only procedural cohesion may contain only part of a complete function or parts of several functions.

Communicational :- A module with communicational cohesion has elements that are related by a reference to the same input or output data. i.e. in a communicationally bound module, the elements are together because they operate on the same input or output data.

Sequential :- When the elements are together in a module because the output of one forms the input to another, we get sequential cohesion. If we have a sequence of elements in which the output of one forms the input to another, sequential cohesion does not provide any guidelines on how to combine them into modules.

Functional :- It is the strongest cohesion. It is functionally bound module, all the elements of the module are related to performing a single function. By functions, we do not mean simply mathematical functions; modules accomplishing a single goal are also included. Functions like "compute square root" and "sort the array" are clear examples of functionally cohesive modules.

Scanned by CamScanner

Function Oriented Design

It is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function.

↳ System is designed from functional view point.

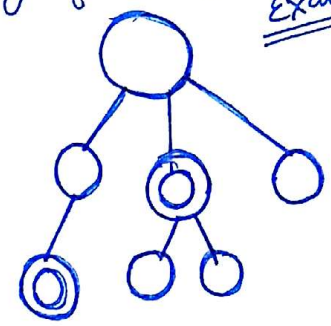
A Generic Procedure

→ Start with a high level description of what the SW/ program does.

→ Refine each part of the description one by one by specifying in greater detail the functionality of each part

⇓ Top Down structure

Example:- Switch-Case

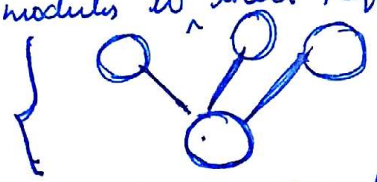


Problem :- Mostly each module is used by atmost 1 other module → parent

Solution :- ↳ Designing of reusable modules

modules used by several other modules to do their required functions

Reusable design



For a function — oriented design, design can be represented mathematically or graphically by using :-

1. DFD : data flow diagram
2. Data dictionary
3. Structure charts
4. Pseudo codes

There are four types of Design strategies

(a) Function oriented Design
(b) Object Oriented Design
(c) Top-Down Design
(d) Bottom-Up Design

Scanned by CamScanner

Object Oriented Software Engineering (OOSE)

→ It is a S/w design technique i.e used in S/w design in Object-Oriented programming.

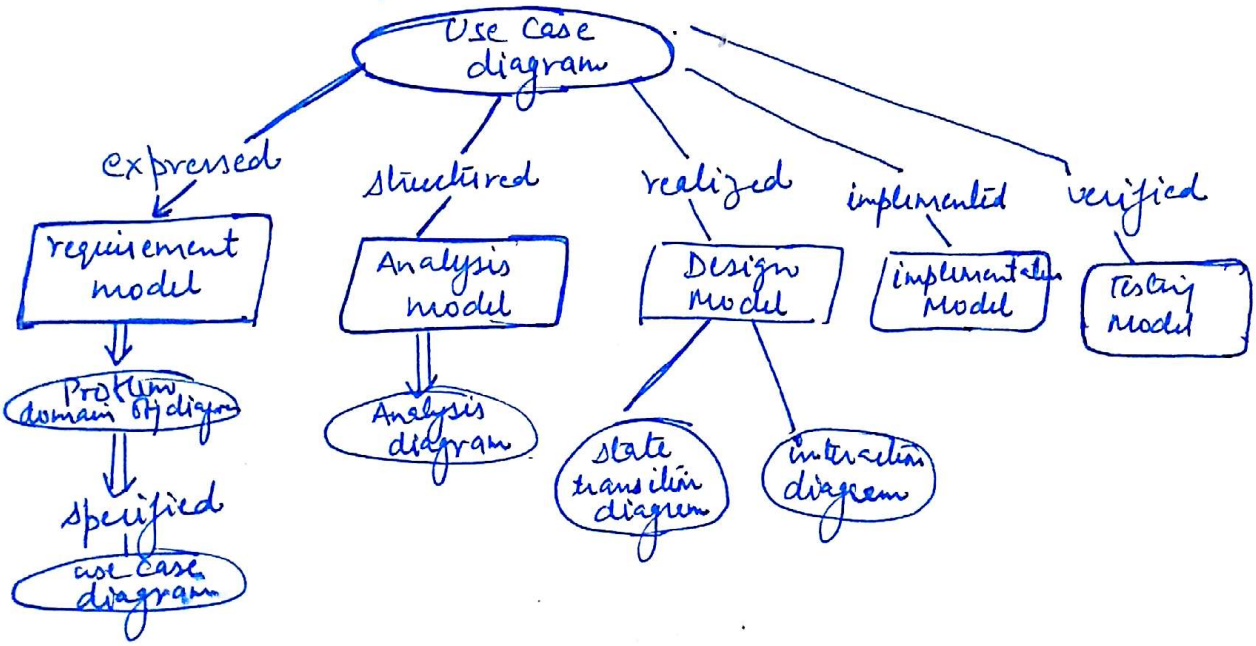
→ OOSE is developed by Ivan Jacobson in 1992.

→ The use of use cases in designing is started by OOSE It help in introducing OO Methodology.

→ It includes a requirements, an analysis, a design an implementation & a testing model

interaction diagram are similar to UML's sequence diagram

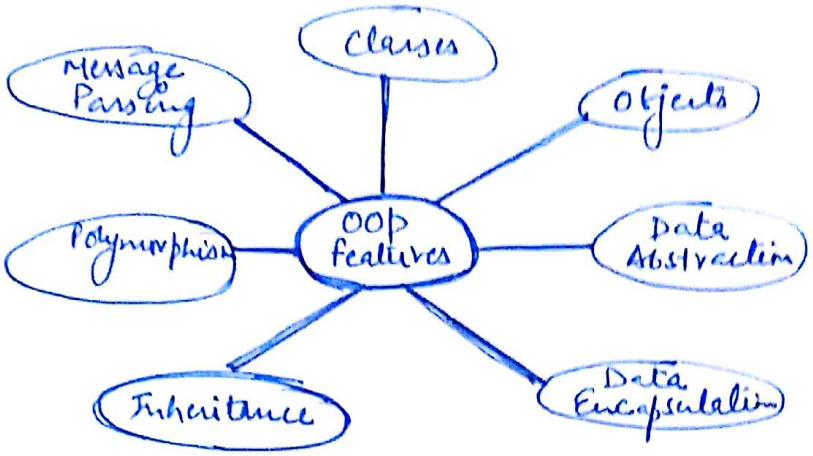
State transition diagram are like UML state chart diagram



Scanned by CamScanner

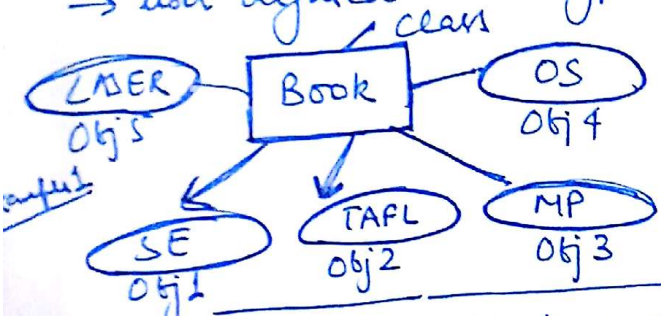
Basic Concept of Object-Oriented

* Objects
* classes
* Data abstraction
* Data Encapsulation
* Inheritance
* Polymorphism
* Message Passing

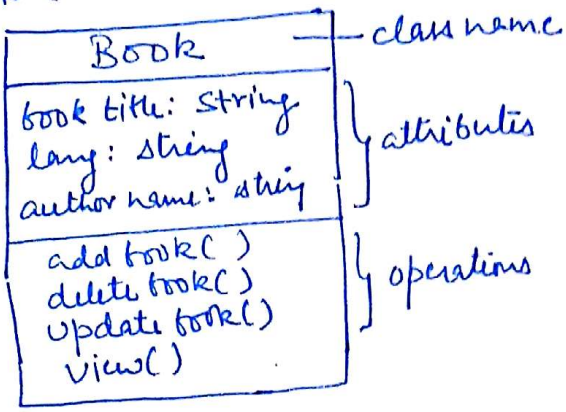


Classes :-

→ user defined data types in which objects are created



Objects with similar properties & methods are grouped together to form a class



Example 2

Objects :-

Noun { run time entities that may represent a person, place, ate in any time

→ Each object contains data & code to manipulate the data

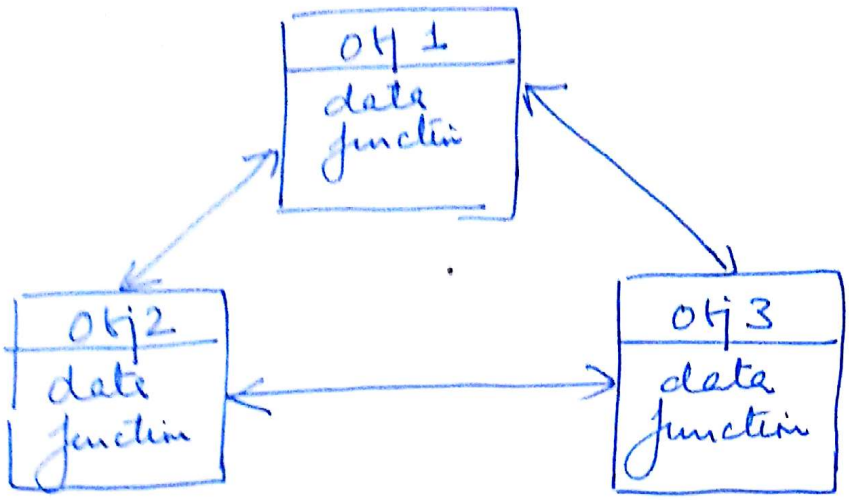
→ data represents the attributes of that object and functions represents the behaviour of that object.

Object — attributes – data
Object — functions – code

→ Object take up space in memory & have an associated add ress structure in C.

→ when a prog. is executed the object interact by sending messages to one another

Scanned by CamScanner



Difference between classes & Objects

Class	Objects
→ class is a datatype	→ Object is an instance of class
→ It generate Objects	→ It gives life to class
→ It doesn't occupy memory location	→ It occupies memory location
→ cannot be manipulated	→ It can be manipulated

~~Example~~:- Terms in Object Oriented Design (OOD) :-

→ **Instance Variables** :- Data, attributes, state that are specific to an instance (Object)

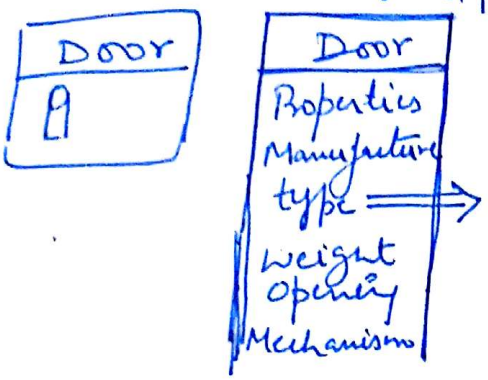
→ **Instance Method** :- operations are specific to an instance (Object)

→ **class variables** :- Data, attributes, state that are non-specific to an instance (Object).

→ **class method** :- operations that are not specific to an instance.

Scanned by CamScanner

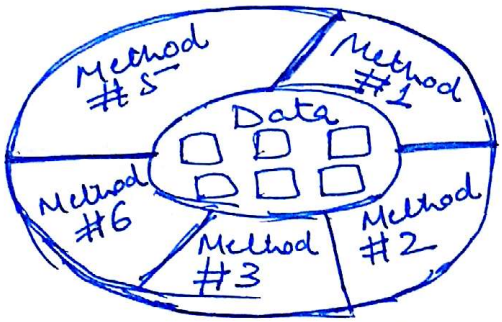
Data abstraction :- It refers to the act of representing essential features W/o including the background details or explanations.



Encapsulation :- The wrapping up of data & functions into a single unit is known as Encapsulation.

→ It is also known as information hiding concept.

→ The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.



Abstraction	Encapsulation
It is used for hiding the unwanted data & giving relevant data about the object class	→ Means hiding the code & data into a single unit to protect data from outside
Outer layout in terms of design	→ how the Object does
Eg:- Outlook of Mobile Phone	→ Inner layout used in term of implementation
	Eg:- Inner implementation details of mobile phone.

Scanned by CamScanner

Polymorphism :- dog → woof, cat → meow, duck → Quack, Person → speak

⇓

"many form"

→ In the real world, the same operation may have different meaning in different situations.

→ The 4 pillars of OO development are
- (a) Encapsulation
- (b) Inheritance
- (c) Polymorphism
- (d) Abstraction

Example:- operation of addition ⇒ 3 ⊕ 4 = 7

⇒ Rama ⊕ Krishna = Ramakrishna

(a) **Operator Overloading :-** The process of making an operator to exhibit difference behaviours in diff. instances

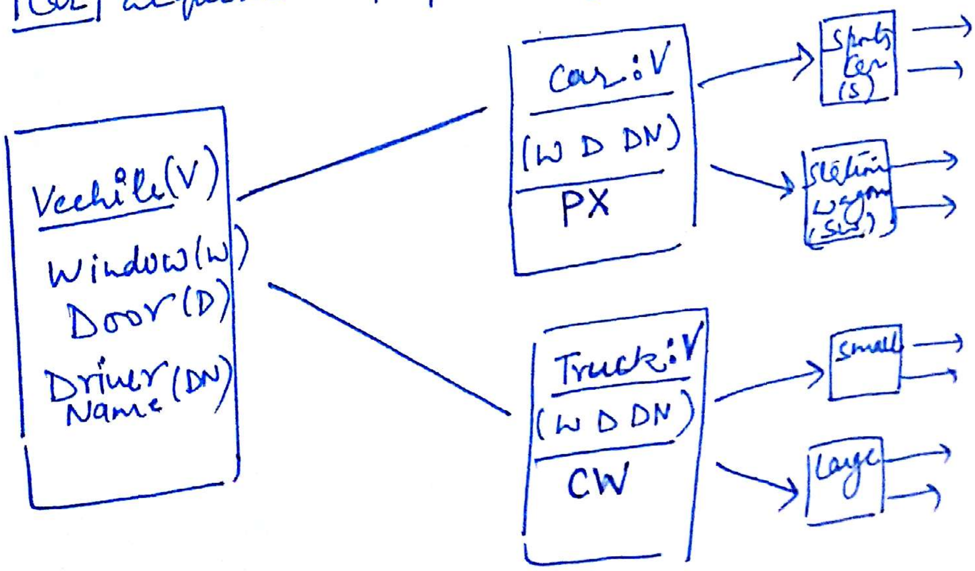
(b) **Function Overloading :-** using a single function name to perform different types of tasks

Inheritance :- is the mechanism by which an object acquires the some / all properties of another object

There is a class [Four wheeler] and its sub class is [Car]   (A) ↔ (B)

So, [Car] acquires the properties of class [Four wheeler]

Example:-



Function Oriented Design (FOD)

1. Focus is on functions
   → Functions → Sub functions
   ... .. more sub functions
2. Emphasis on "what the system does"
   VERB
3. Example :- ~~ATM~~ ~~Machine~~ Banking S/S
   Withdraw, Deposit, Transfer

```
Withdraw()
{ Defn... }
Deposit()
{ Defn... }
Transfer()
{ Defn.. }
```

Methodology is SDLC
Key focus is functions
Risk is very high
Reusability is low
Suitable for well defined projects with stable user requirements

Object Oriented Design (OOD)

focus is on objects
→ Parent Object → child Object
... etc.

Emphasis on "what system consists of ?"
NOUN

Example :- Banking S/S
Customer, Money, Account

Methodology is Incremental
Key focus is objects
Risk is very low
Reusability is high
For project with changing user requirements

Scanned by CamScanner

Top-Down and Bottom-Up Strategies

Top-Down

1) A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved

2) Top-down design methods often result in some form of stepwise refinement.

3) A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. Hence, it is a ~~poor~~ reasonable approach if a waterfall type of process model is being used.

Bottom-Up

1) A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components.

2) Bottom-up methods work with layers of abstraction.

3) A bottom-up approach is suitable, if a system is to be built from an existing system, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable.

Scanned by CamScanner

Software Measurement and Metrics

(a) **Measure** :- quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.
E.g :- Number of Errors.

(b) **Metric** :- quantitative measure of degree to which a system, component or process possesses a given attribute.
"A handle or guess about a given attribute"
E.g :- Number of Errors found per person hours expended.

Why Measure Software?

→ Determine the quality of the current product or process.
→ Predict qualities of a product / process.
→ Improve quality of a product / process.

Motivation for Metrics

→ Estimate the cost & schedule of future projects
→ Evaluate the productivity impacts of new tools and techniques.
→ Establish productivity trends over time
→ Improve software quality
→ Forecast future staffing needs
→ Anticipate and reduce future maintenance needs.

Example :-

(a) Defect rates
(b) Error rates
(c) Measured by
  — individual
  — module
  — during development
(d) Errors should be categorized by origin, type, cost.

Scanned by CamScanner

Metric classification

→ Products
- Explicit results of software development activities
- Deliverables, documentation, by products

→ Processes
- Activities related to production of software

→ Resources
- Inputs into the software development activities
- hardware, knowledge, people

Product Vs Process

→ Process Metrics
- Insights of process paradigm, software Engineering tasks, work product, or milestones
- Lead to long term process improvement

→ Product Metrics
- Assesses the state of the project
- Track potential risks
- Uncover problem areas
- Adjust workflow or tasks
- Evaluate teams ability to control quality

Types of Measures

→ Direct Measures (internal attributes)
- Cost, effort, LOC, Speed, memory

→ Indirect Measures (external attributes)
- functionality, quality, complexity, efficiency, reliability, maintainability.

Scanned by CamScanner

Size-Oriented Metrics

→ Size of the software produced
→ LOC – Lines of Code
→ KLOC – 1000 Lines of Code
→ SLOC – statement Lines of Code
→ Typical Measures :- (a) Errors/KLOC
(b) Defects/KLOC
(c) Cost/LOC
(d) Documentation Pages/KLOC

LOC Metrics

→ Easy to use
→ Easy to compute
→ Language & programmer dependent

Complexity Metrics

→ LOC – a function of complexity
→ Language & programmer dependent
→ Halstead's Software Science (Entropy Measures)
→ n_1 : number of distinct operators
→ n_2 : number of distinct operands
→ N_1 : total no. of operators
→ N_2 : total no. of operands

Example:-

```
if( K<2 )
{ if( K>3)
    X = X * K;
}
```

Distinct operators : if ( ) { } > < = * ;
Distinct operands : K 2 3 X

$n_1 = 10$
$n_2 = 4$
$N_1 = 13$
$N_2 = 7$

Scanned by CamScanner

HALSTEAD'S METRICS

```
if (K<2)
{ K=3;
  X=X*K;
}
```

operators: if( ) { } = ; * + -

operands: X K 3,2,4 etc

$n \rightarrow$ Distinct
$N \rightarrow$ Total
$1 \rightarrow$ operators
$2 \rightarrow$ operands

n_1 n_2 N_1 N_2

1. $N = N_1 + N_2$ ↓ Length of program
2. $n = n_1 + n_2$ ↓ Vocabulary of program
3. Purity Ratio $= \frac{\hat{N}}{N}$

$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$ ↓ Estimated Length

4. Volume (V) $= N \log_2 n$
5. Difficulty (D) $= \frac{n_1}{2} \times \frac{N_2}{n_2}$
6. Effort $= D * V$

Ques:- $N, n, \hat{N}/N, V, D, E$

```
if (X>5)
{ X=X+2;
  if (X<7)
  { X=0;
  }
}
```

N_1 = if (15) ( ) > < = ; + if ( ) < > = ; { }

N_2 = (9) x 5 x x 2 x 7 x 0

$n_1 = 9$

$n_2 = 5$

① Total Length (N) $= N_1 + N_2 = 15 + 9 = 24$

② Vocabulary (n) $= n_1 + n_2 = 9 + 5 = 14$

③ $\frac{\hat{N}}{N}$ $\quad \hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$

$= 9 \log_2 9 + 5 \log_2 5$

$= 9 \frac{\log_{10} 9}{0.3010} + 5 \frac{\log_{10} 5}{0.3010}$

$= 9 \times (3.17) + 5 \times (2.32)$

$= 28.53 + 11.6$

$= 40.13$

$\frac{\hat{N}}{N} = \frac{40.13}{24} = 1.67$

Scanned by CamScanner

④ $V = N \log_2 n$

$= 24 \log_2 14$

$= 24 \times \frac{\log_{10} 14}{0.3010}$

$= 24 \times \frac{\log_{10} 14}{\log_{10} 2}$

$= 24 \times 3.807$

$= 91.36$, $\boxed{V = 91.36}$

⑤ $D = \frac{w_1}{2} \times \frac{N_2}{w_2}$ (2c)

$= \frac{9}{2} \times \frac{9}{5} = \frac{81}{10} = 8.1$

$\boxed{D = 8.1}$

⑥ $E = D \times V$

$= 8.1 \times 91.36 = 740.016$

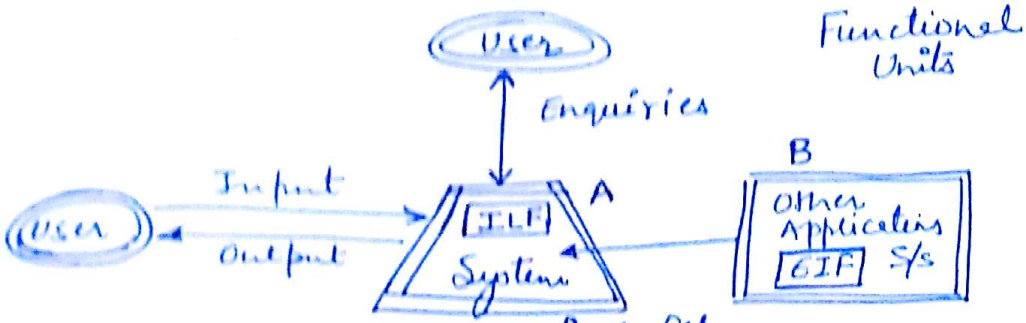
$\boxed{E = 740.016}$

Scanned by CamScanner

Software Size Oriented Measures
Function Point (FP) Based Measures

→ It measures functionality from user's point of view
→ What the user receives from SW + what the user requests from SW.

→ Focuses on what functionality is being delivered.



ILF : Internal Logical Files
EIF : External Interfaces

A system has 5 types of functional Units :-

Internal Logical Files (ILF): → Data function Types
- The control information or logically related data i.e. stored WITHIN the system.

External Interfaces files (EIF): → Data function Types
- The control data, or other logical data i.e. referenced by the system but present in another system.

External Inputs (EI): The data/control info that comes from outside our system.

External Outputs (EO): The data that goes out of the system after generation.

External Enquiries (EQ): Combination of I/P – O/P resulting

Transaction function Types

Scanned by CamScanner

COUNTING FUNCTION POINTS

Step 1: Each function point is ranked according to complexity (Low, Average, High). There exists pre-defined weights for each F.P. in each category. (Table A)

Functional Units		weighing factors		
		Low	Average	High
Individual Units	EI	3	4	6
	EO	4	5	7
	EQ	3	4	6
	ILF	7	10	15
	EIF	5	7	10

Step 2: Calculate Unadjusted function point by multiplying each F.P. by its corresponding weight factor

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} (Zij * Wij)$$

- Zij → count of functional units of category i classified as complexity j
- Wij → weight from Table A

How to assign weights or rank function points?
↳ Dependent on the organization
↳ Based on past projects

Step 3: Calculate Final function Points

$$\text{Final F.P.} = UFP \times CAF$$

CAF → Complexity Adjustment factor

Calculated using 14 aspects of processing complexity questions answered on a scale of 0 to 5.

0 → No. Influences
1 → Incidental
2 → Moderate
3 → Average
4 → Significant
5 → Essential

$$CAF = [0.65 + 0.01 \times \Sigma fi]$$

(i) → varies from 1 to 14.

Scanned by CamScanner

Advantages of Function Point Approach :-

* Size of sw delivered is measured independent of
  - → language
  - → technology & tools
* Function Point (FP) directly estimated from requirements, before design & coding
  - → we get an estimate of sw size even before major design or coding happens (early phases)
  - → Any change in requirements can be easily reflected in FP count.
* Useful even for those users w/o technical expertise
  - → F.P. is based on external structure of the s/w to developed.
  - →

Example:- Given the following values, compute F.P. when all complexity adjustment factors and weighting factors are average

$\sum_{i=1}^{14} f_i \rightarrow 14 \times 3$

User I/p = 50
User O/p = 40
User Enquiries = 35
User files = 6
External Interfaces = 4

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} z_{ij} w_{ij}$$

$$UFP = 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7$$
$$= 200 + 200 + 140 + 60 + 28$$
$$= 628$$

$$CAF = 0.65 + (0.01 \times \Sigma f_i) = 0.65 + 0.01 (14 \times 3)$$
$$= 0.65 + 0.42$$
$$= 1.07$$

$$FP = UFP \times CAF$$
$$= 628 \times 1.07$$

Scanned by CamScanner

Cyclomatic Complexity Measures: Control Flow Graphs

→ It is a software metric used to measure the complexity of a program. It was developed by Thomas J. McCabe, Sr in 1976.

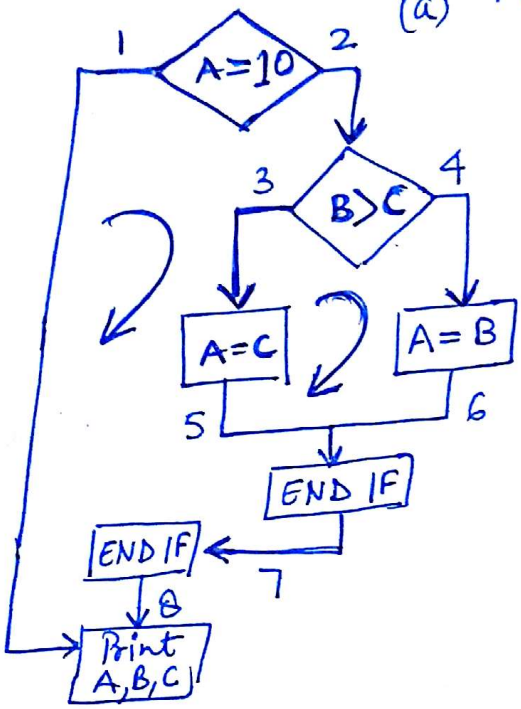
Formula: (a) Cyclomatic Complexity (M) ⇒ E − N + P
(E: No. of Edges, N: No. of Nodes, P: Predicate Nodes)

(b) M = R + 1 where R is a Region

Example 1:- Algorithm

```
If A=10 Then
  If B>C Then
    A=B
  ELSE
    A=C
  ENDIF
ENDIF
Print A
Print B
Print C
```

Flow graph



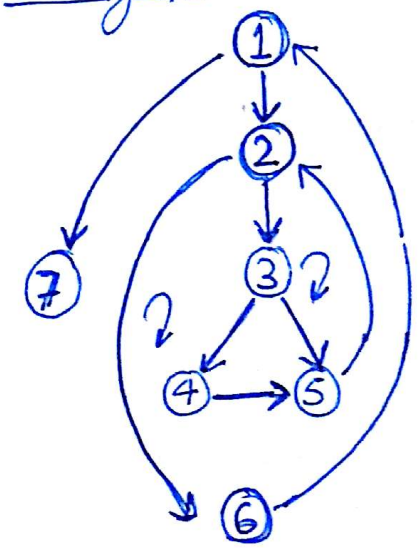
(a) M = E − N + P
E = 8
N = 7
P = 1
⇒ 8 − 7 + 2
= 3

(b) M = R + 1
= 2 + 1
= 3

Example 2:-

```
i=0
n=4, // nodes
while (i<n-1) do
    j=i+1;
while (j<n) do
  if A[i] < A[j] then
     Swap(A[i], A[j])
end do,
  i=i+1;
end do.
```

Flow graph



(a) E = 9, N = 7, P = 1
⇒ 9 − 7 + 2 = 4

(b) R = 3
M = 3 + 1 = 4

Scanned by CamScanner

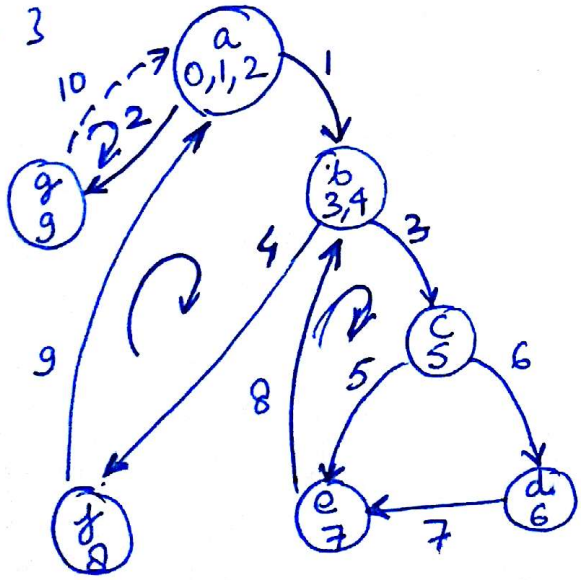
Example 3:

```
int BinSearch ( char *item, char *table[], int x)
{  int bot = 0;
   int top = n-1;
   int mid, cmp;
✓while ( bot <= top)
{  mid = ( bot + top)/2;
✓if ( table [mid] == item)
     return mid;
✓else if ( compare (table [mid], item) < 0 )
     top = mid-1;
   else
     bot = mid+1; }
   return -1; }
```

* if, else if, switch case, for, while, do-while } 1 for each case and add 1
(Shortcut for finding cyclomatic complexity)

Example 4:-

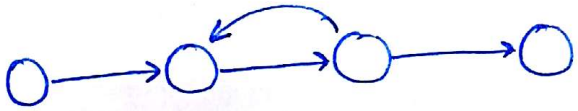
```
0 {
1.  i = 1
2. ✓while (i<=n) {
3.     j=1;
4. ✓while( j<=i){
5.  ✓if ( A[i] < A[j])
6.     swap (A[i], A[j]);
7.    j = j+1; }
8.    i = i+1; }
9. }
```



M = 10 - 7 + 1 = 4
M = 3 + 1 = 4

Example 5:-

(i) 

(ii) 

(iii) 

Calculate the cyclomatic complexity of three parts of Example 5.

Scanned by CamScanner