UNIT -III

Establishing Testing Policy

A software testing policy serves two purposes.

- **First,** it is the basis for defining what software testers will include in the **test processes.**
- **Second,** it explains to outside parties, such as organizational management, IT customers and users, as well as project person-nel, the **role and responsibilities of software testing.**

- ✓ A Test Policy is a high level document and is at the top of the hierarchy of the Test Documentation structure.

- ✓ The purpose of the Test Policy document is to represent the ***testing philosophy of the company as a whole and to provide a direction which the testing department should adhere to and follow.***

- ✓ It should apply to both new projects and maintenance work**.**

Establishing Testing Policy

Criteria for a Testing Policy

- **A testing policy is management's definition of testing for a department .**
- **A testing policy involves the following four criteria:**

1. **Definition of testing**. A brief but clear definition of testing.
2. **Testing system**. The method through which testing will be achieved and enforced.
3. **Evaluation**. How information services management will measure and evaluate testing.
4. **Standards.** The standards against which testing will be measured.

Establishing Testing Policy

TESTING POLICY
ABC INFORMATION TECHNOLOGY DEPARTMENT

TESTING DEFINITION

Determine the validity of the computer solution to a business problem.

TESTING SYSTEM

Develop and execute a test plan in accordance with departmental procedures and user requirements.

MEASUREMENT OF TESTING

Calculate the cost of correcting defects not discovered during testing.

TESTING STANDARDS

Allow only one defect per 250 executable program statements.

Philip Jones
George Wilson
Elizabeth Charney
Max Hartman

Methods for Establishing a Testing Policy

- **The following three methods can be used to establish a testing policy:**

1. Management directive.

- One or more senior IT managers write the policy.
- They determine what they want from testing, document that into a policy, and issue it to the department.
- This is an economical and effective method to write a testing policy
- the potential disadvantage is that it is not an organizational policy, but rather the policy of IT manageme**nt.**

Methods for Establishing a Testing Policy

2. Information services consensus policy.

- IT management convenes a group of the more senior and respected individuals in the department to jointly develop a policy.
- While senior management must have the responsibility for accepting and issuing the policy, the development of the policy is representative of the thinking of all the IT department, rather than just senior management.
- The advantage of this approach is that it involves the key members of the IT department. Because of this participation, staff is encouraged to follow the policy.
- The disadvantage is that it is an IT policy and not an organizational policy.

Methods for Establishing a Testing Policy

3. Users' meeting.

- Key members of user management meet in conjunction with the IT department to jointly develop a testing policy.
- Again, IT management has the final responsibility for the policy, but the actual policy is developed using people from all major areas of the organization.
- The advantage of this approach is that it is a true organizational policy and involves all of those areas with an interest in testing.
- The disadvantage is that it takes time to follow this approach, and a policy might be developed that the IT department is obligated to accept because it is a consensus policy and not the type of policy that IT itself would have written.

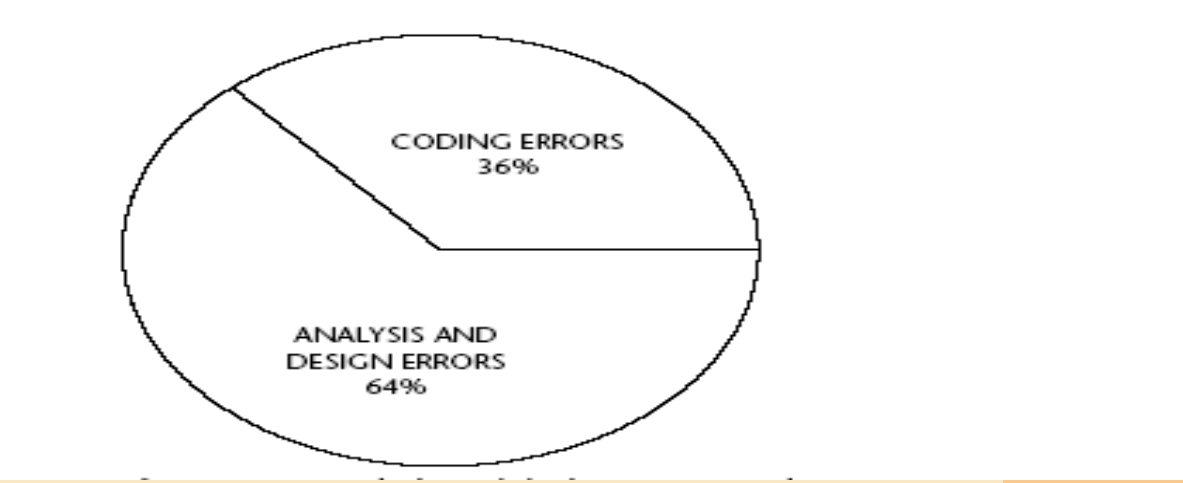
Structured Approach to Testing

- All too often, testing after coding is the only verification technique used to determine the adequacy of the system.
- All errors are costly, but the later in the SDLC the error is discovered, the more costly the error. Indeed, an error discovered in the latter parts of the SDLC must be paid four different times.
  - The first cost is developing the program erroneously, which may include writing the wrong specifications, coding the system wrong, an documenting the system improperly.
  - Second, the system must be tested to detect the error.
  - Third, the wrong specifications and coding must be removed and the proper specifications, coding, and documentation added.
  - Fourth, the system must be retested to determine whether the problem(s) have been corrected.

Structured Approach to Testing

- ✓ If information services has as its goal lower cost and higher quality systems
- ✓ It must not isolate verification to a single phase in the development process; rather, it must incorporate verification into each phase of development.
- ✓ One of the most prevalent and costly mistakes on systems development projects today is to defer the activity of detecting and correcting problems until late in the project.
- ✓ A major justification for an early verification activity is that many costly errors are made before coding begins.

Structured Approach to Testing



- Studies have shown that the majority of system errors occur in the design phase.
- Figure represents the results of numerous studies that show that approximately two-thirds of all detected system errors can be attributed to errors made during the design phase.
- This means that almost two-thirds of the errors must be specified and coded into programs before they can be detected

Structured Approach to Testing

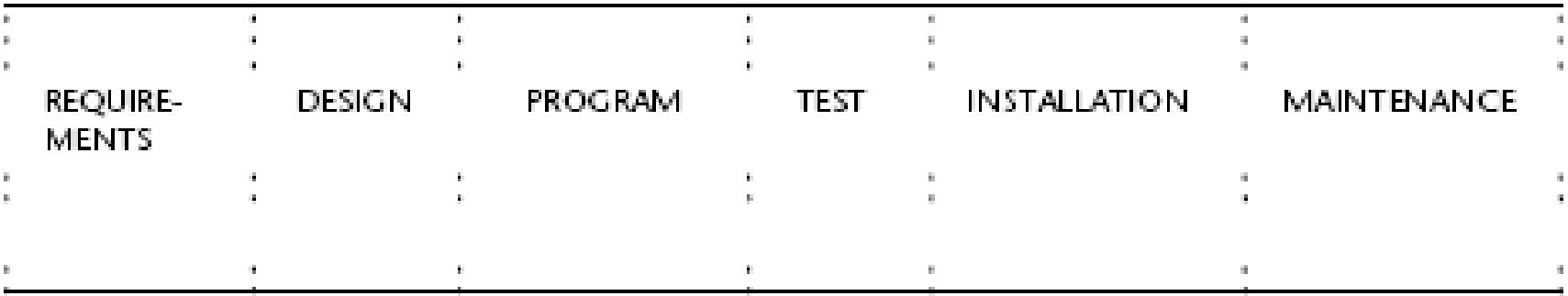


Figure 2-6 Traditional software development life cycle.

The following activities should be performed at each phase:

■ Analyze the software documentation for internal testability and adequacy.

■Generate test sets based on the software documentation at this phase.

In addition, the following should be performed during the design and program phases:

■ Determine that the software documentation is consistent with the software documentation produced during previous phases.

■Refine or redefine test sets generated earlier.

Structured Approach to Testing

Table 2-1 Life Cycle Verification Activities

LIFE CYCLE PHASE	VERIFICATION ACTIVITIES
Requirements	■ Determine verification approach ■ Determine adequacy of requirements ■ Generate functional test data ■ Determine consistency of design with requirements
Design	■ Determine adequacy of design ■ Generate structural and functional test data ■ Determine consistency with design
Program	■ Determine adequacy of implementation ■ Generate structural and functional test data for programs
Test	■ Test application system
Installation	■ Place tested system into production
Maintenance	■ Modify and retest

Structured Approach to Testing

Requirements

- The verification activities performed during the requirements phase of software development are extremely significant.
- The adequacy of the requirements must be thoroughly analyzed and initial test cases generated with the expected (correct) responses.
- Developing scenarios of expected system use may help to determine the test data and anticipated results.
- Requirements defined to later phases of development can be very costly.
- A determination of the importance of software quality attributes should be made at this stage.
- Both product requirements and validation requirements should be established.

Structured Approach to Testing

Design

- During the design phase, the general testing strategy is formulated and a test plan is produced.
- If needed, an independent test team is organized.
- A test schedule with observable milestones should be determined.
- At this same time, the framework for test documentation should be established.
- During the design phase, validation support tools should be acquired or developed and the test procedures themselves should be produced.
- In addition to test organization and the generation of test cases, the design itself should be analyzed and examined for errors.

Structured Approach to Testing

- Simulation can be used to verify properties of the system structures and subsystem interaction, design walkthroughs should be used by the developers to verify the flow and logical structure of the system, while design inspection should be performed by the test team.
- Areas of concern include missing cases, faulty logic, module interface mismatches, data structure inconsistencies, erroneous I/O assumptions, and user interface inadequacies.
- The detailed design must prove to be internally coherent, complete, and consistent with the preliminary design and requirements.

Structured Approach to Testing

Program

- Actual testing occurs during the program stage of development. Many testing tools and techniques exist for this stage of system development.
- Code walkthrough and code inspection are effective manual techniques.
- Static analysis techniques detect errors by analyzing program characteristics such as data flow and language construct usage.
- For programs of significant size, automated tools are required to perform this analysis. Dynamic analysis, performed as the code actually executes, is used to determine test coverage through various instrumentation techniques.
- Formal verification or proof techniques are used to provide further quality assurance.

Structured Approach to Testing

Test

- During the test process, Test sets, test results, and test reports should be catalogued and stored in a database.
- For all but very small systems, automated tools are required to do an adequate job
- test data generation aids, test coverage tools, test results management aids, and report generators are usually required.

Installation

- The process of placing tested programs into production is an important phase normally executed within a narrow time span.

Structured Approach to Testing

- Testing during this phase must ensure that the correct versions of the program are placed into production, that data if changed or added is correct, and that all involved parties know their new duties and can perform them correctly.

Maintenance

- More than 50 percent of a software system's life cycle costs are spent on maintenance.
- As the system is used, it is modified either to correct errors or to augment the original system. After each modification, the system must be retested.
- Such retesting activity is termed *regression testing. The goal of regression testing is to minimize the cost of system* revalidation.

Structured Approach to Testing

- Usually only those portions of the system impacted by the modifications are retested. However, changes at any level may necessitate retesting, re-verifying, and updating documentation at all levels below it.
- For example, a design change requires design re-verification, unit retesting, and subsystem retesting.
- Test cases generated during system development are reused or used after appropriate modifications.

Economics of System Development Life Cycle testing

- The risk of over-testing is the unnecessary use of valuable resources in testing computer systems that have no flaws, or so few flaws that the cost of testing far exceeds the value of detecting the system defects.
- Effective testing is conducted throughout the system development life cycle (SDLC).
- The SDLC represents the activities that must occur to build software, and the sequence in which those activities must occur.

Economics of System Development Life Cycle

Most of the problems associated with testing occur from one of the following causes.

- Failing to give the testing objective.
- Testing at the wrong phase in the life cycle
- Using ineffective testing techniques

❖The cost-effectiveness of testing is illustrated in Figure as a testing cost curve.

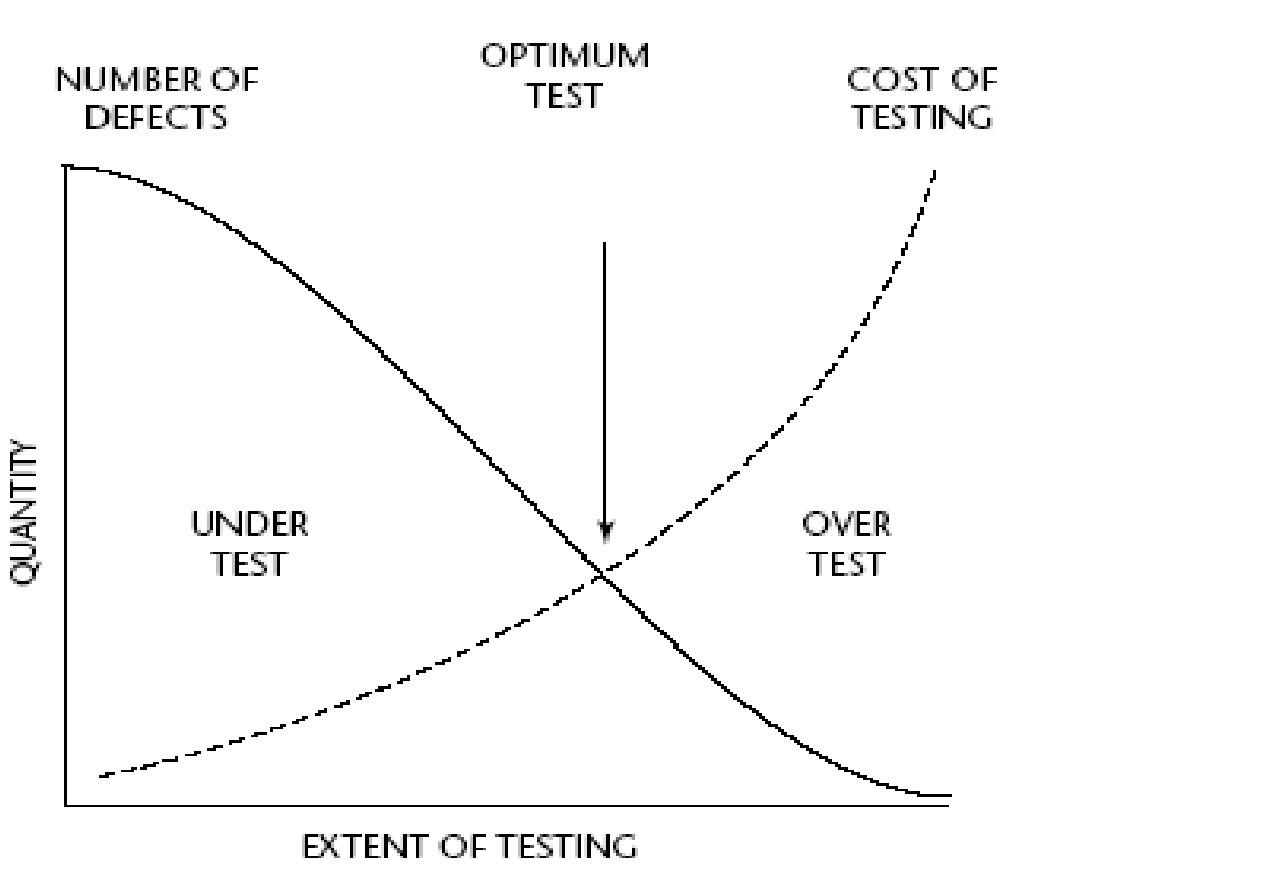
❖As the cost of testing increases, the number of undetected defects decreases.

❖The left side of the illustration represents an under-test situation in which the cost of testing is less than the resultant loss from undetected defects.

❖At some point, the two lines cross and an over-test condition begins. In this situation, the cost of testing to uncover defects exceeds the losses from those defects.

.

Economics of System Development Life Cycle testing



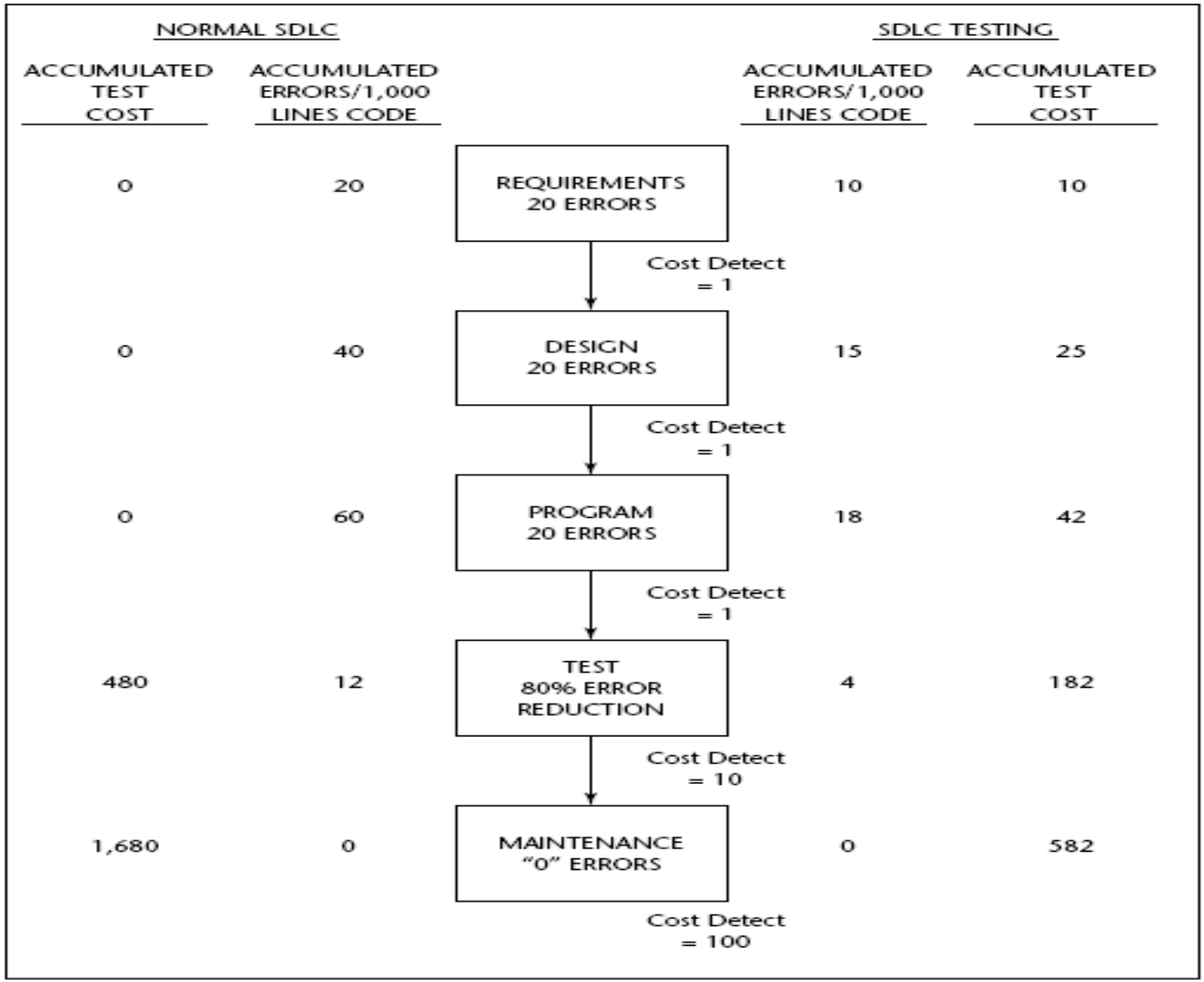
Testing Cost Curve

Economics of System Development Life Cycle testing

- These facts are illustrated in Figure for a hypothetical system with 1,000 lines of source code. A normal SDLC test process is shown on the left in which testing occurs only after coding.
- In this example, all 60 errors remain after coding for testing, which detects 48 errors (60 times 80 percent equals 48) at an average cost ten times as great as those detected prior to coding, resulting in a cost of 480 units.
- To that, we must add 1,200 units of cost representing the 12 remaining errors to be detected during production at a cost of 100 units each.
- The net test cost is 1,680 units. Using life-cycle testing,
- this can be reduced to 582 units or only one-third of the normal SDLC test concept cost (illustrated on the right side of Figure 2-5).

Economics of System Development Life Cycle testing

ECONoMICs of SDLC Testing



Software Quality

McCall's quality factors

Factors that affect software quality can be categorized in two broad groups:

1. Factors that can be directly measured (e.g. defects uncovered during testing)

2. Factors that can be measured only indirectly (e.g. usability or maintainability)

Product metrics

- Product metrics for computer software helps us to assess quality.
- **Measure**
  - When you obtain/observe/measure a value of directly observable property of an entity, you have a measure
  - . Each measure has a standard unit of measure (UOM) like seconds, meter, kilograms etc.
  - In software development and software testing, most commonly used measures are:
  - •Number of Defects found in a system or component
  - •Lines of Code (LOC, kLOC)
  - • Number of Test Cases

Product metrics

- **Metric**

A metric in contrast, is a derived value which cannot be observed/measured directly.

- It is a number derived from one or more measures by a formula (or estimation).
- Best known metrics in software development and software testing are:
- •Number of defects found per kLOC, which serves as an estimation of quality of code
- • Productivity, i.e. Size / Effort

- **Indicator**

-- An indicator is “a thing that indicates the state or level of something”, thus it can be simply just a number showing value of a particular measure or metric.

A metric or a combination of metrics that provide insight into the software process, a software project or a product itself

Product Metrics for analysis , Design , Test and maintenance

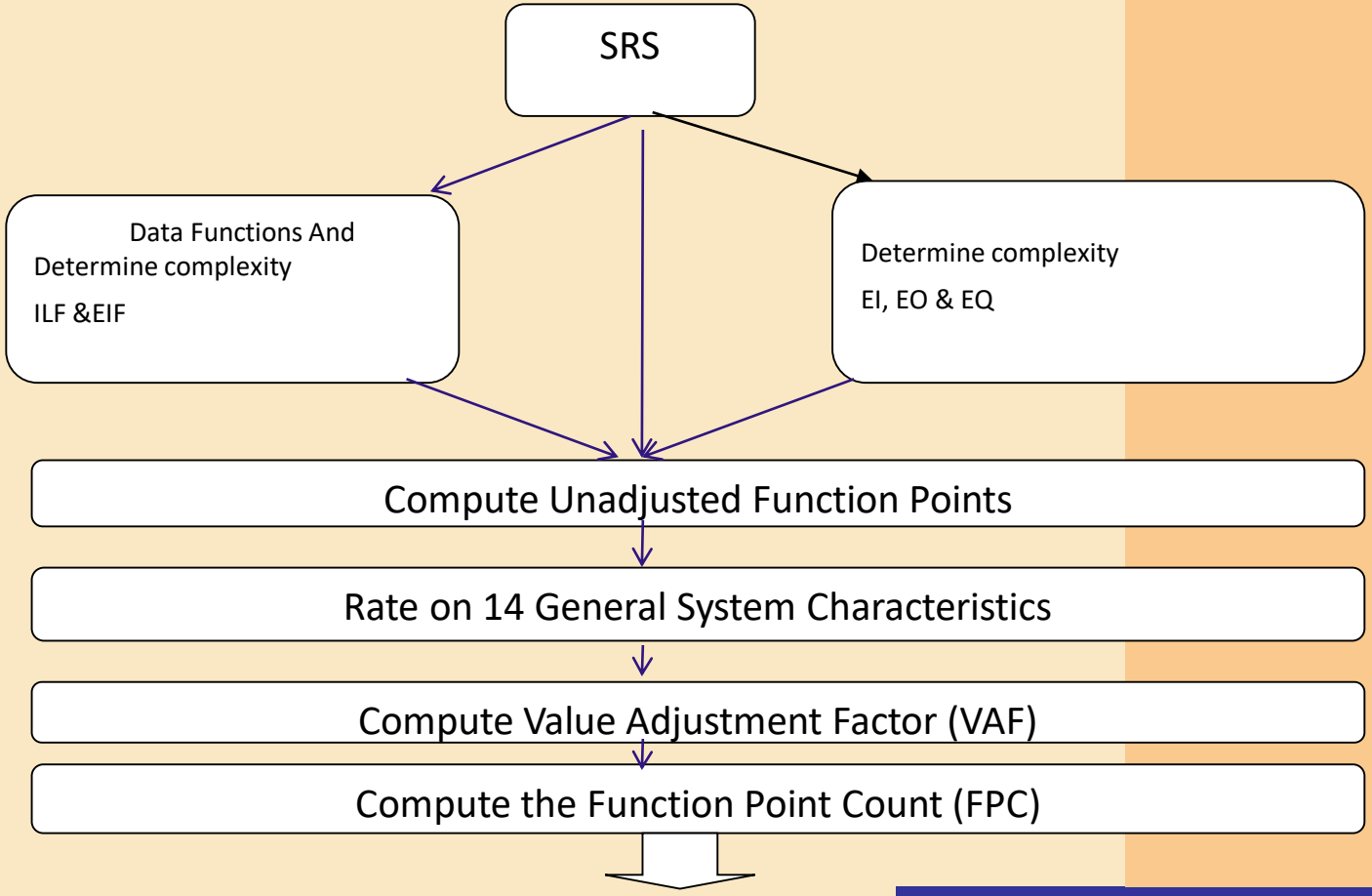
- Product metrics for the Analysis model
  - Function Point Analysis (FPA)
  - FPA is a popular method for estimating and measuring the size of application software based on the functionality of the software from the user's point of view.
  - Thorough this method, the size of an application system's functionality is calculated in terms of Function Point Count (FPC).
  - Allan J Albrecht of IBM proposed FPC as a size measure in 1970's.

Product metrics for the Analysis model

- According to IFPUG, the objective of function point analysis are to:
  - Measure functionality of a software system as seen from the user's perspective.
  - Measure the size of software systems independent of technology used for implementation.
  - Create a measurement methodology that is simple enough to minimize the overhead of the measurement process.
  - Create a consistent measure among various projects and organizations.
- Function point can be computed for a new system being developed (called Development Function Point) as well as for Enhancement work (called Enhancement Function Points).

Product metrics for the Analysis model

Counting Development Function Points



- The quality of the SRS, which is an output of the requirement definition, is critical to the accurate determination of the size of the system in terms of FPC.
- In the first two steps we identify the various functions in the system.
- From the FPA point of view, function could be either data functions or transaction functions.
- Only user requested and user identifiable functions are identified.
- Data functions relate to the functionality of the system
- Transaction functions contribute functionality by taking in data or extracting and providing data from system to users or other systems.
- Data and transaction functions can be extracted from the functional requirements of the SRS.

- Unadjusted function points are assigned for all identified functions, to arrive at the total unadjusted function point.
- In the next step, we rate the system on 14 general system characteristics (GSC).
- Using a formula we compute what is called a value adjusted factor (VAF).
- Multiply the VAF with the unadjusted function point to arrive at the function point count (FPC).

Identify Data Functions and Determine their complexity

- Data function types represent functionality provided to the user to meet internal and external data representations.
- Two data function types
  - Internal Logical Files(ILF) and
  - External Interface Files (EIF).

- An Internal Logical Files (ILF) is a group of information stored in the system and maintained by the system.
- ILF is a logical grouping of data or control information that is stored in the system, used within it and maintained by it.
- External Interface Files (EIF) is a unique file that is shared between the system and some external application(s).
- It is a user identifiable logical grouping of data or control information that is used by the system, but not maintained by it.

- To determine the complexity of ILFs and EIFs, the number of Data Element Types (DET) and Record Element Types (RET) in each ILF and EIF are identified.
- A DET is a user recognizable unique and non-recursive field in the ILF/EIF.
- A user recognizable sub group of DETs within an ILF/EIF is an RET (also called data structure, logical group of fields).
- The number of DETs and RETs determine the complexity of an ILF or EIF.
- The higher the number of DETs or RETs, the higher is the complexity.

➓ The complexity is determine using the following table:

	1-19 DETs	20-50 DETs	>51 DETs
1 RET	simple	simple	average
2-5 RET	simple	average	complex
>6 RET	average	complex	complex

➢Identify Transaction Functions and Determine their complexity:

•Transaction function types represent the functionality provided to the user to process data by the application and comprise EI, EO and EQ.

•**External Input (EI):** EI is a unique business event that needs to update the data in system.

•External Output (EO): is a unique data or control output that crosses the boundary to go out of a system. Typically, reports would be identified as EOs:

•External Inquiry (EQ): is a unique combination of an input and an output where the input causes the immediate generations of the output.

•To determine the complexity of an EI, the number of DET and File Types Referenced (FTR) are identified.

•The complexity of the EQ is determined using the following table:

	1-5 DETs	**6-19 DETs**	**>20 DETs**
0-1 FTR	simple	simple	average
2-3 FTR	simple	average	complex
>4 FTR	average	complex	complex

Compute unadjusted function points:

- The unadjusted Function Points for each function depends on the function type (ILF, EIF, EI, EO, and EQ) complexity of the function (simple, average and complex) determined in the previous section.
- At the end of this step, the column “UFP” should be filled and the “Total UFP” in the IKF, EIF, EI, EO, and EQ tables should be filled up.

Complexity	Function Type				
	ILF	EIF	EI	EO	EQ
Simple	7	5	3	4	3
Average	10	7	4	5	4
Complex	15	10	6	7	6

Rate on 14 General System Characteristics

- The FPA method recognizes 14 GSCs that influence the Function Point Count (FPC).
- These GSCs are related to the “non-functional requirements” and “design constraints” identified in the SRS.
- Each GSCs associated descriptions that help determine the degree of influence of that characteristic.
- The rating for the degree of influence for each characteristics ranges on a scale of zero to five, from no influence to strong influence.

Compute Value Adjustment Factor (VAF):

Value Adjustment Factor calculation			
S.No	GSC Name	Rating	Comments/reasons
1	Data communication		
2	Distributed data processing		
3	Performance		
4	Heavily used configuration		
5	Transaction rate		
6	On-line data entry		
7	End user efficiency		
8	Online update		
9	Complex processing		
10	Reusability		
11	Installation ease		
12	Operational ease		
13	Multiple sites		
14	Facilitate change		
	Total (TDI) =		
	VAF = (TDI * 0.01)+0.65 =		

- The next step is to compute the VAF.
- This is uses 14 GSCs and the rating assigned to each of them in the previous step.
- The degrees of influence rating of each GSC are then added to give the Total Degree of Influence (TDI) is then computed using the formula.
- VAF = (TDI*0.01) +0.65
- The value of TDI ranges from a minimum of 0 to a maximum 70.

Value Adjustment Factors

1) Does the system require reliable backup and recovery?
2) Are specialized data communications required to transfer information to or from the application?
3) Are there distributed processing functions?
4) Is performance critical?
5) Will the system run in an existing, heavily utilized operational environment?
6) Does the system require on-line data entry?
7) Does the on-line data entry require the input transaction to be built over multiple screens or operations?

(More on next slide)

Value Adjustment Factors (continued)

8) Are the internal logical files updated on-line?

9) Are the inputs, outputs, files, or inquiries complex?

10) Is the internal processing complex?

11) Is the code designed to be reusable?

12) Are conversion and installation included in the design?

13) Is the system designed for multiple installations in different organizations?

14) Is the application designed to facilitate change and for ease of use by the user?

Compute the Function Point Count (FPC):

- For a development project the function point is computed as

$$FPC = UFP * VAF$$

FUNCTION POINTS ESTIMATION FORM

1.1 Determine Unadjusted Function Point Count

	Measurement Parameter	Count		Weighting Factor Low	Average	High		Total
1.	External Inputs		x	3	4	6	=	
2.	External Outputs		x	4	5	7	=	
3.	External Inquiries		x	3	4	6	=	
4.	Internal Logical Files		x	7	10	15	=	
5.	External Interface Files		x	5	7	10	=	
	Unadjusted Function Point Total →							

1.2 Determine Value Adjustment Factor

Rate Each Factor: (0 - No Influence, 1 - Incidental, 2 - Moderate, 3 - Average, 4 - Significant, 5 - Essential)

1.	How many data communication facilities are there?	
2.	How are distributed data and processing functions handled?	
3.	Was response time or throughput required by the user?	
4.	How heavily used is the current hardware platform?	
5.	How frequently are transactions executed?	
6.	What percentage of the information is entered online?	
7.	Was the application designed for end-user efficiency?	
8.	How many internal logical files are updated by on-line transaction?	
9.	Does the application have extensive logical or math processing?	
10.	Was the application developed to meet one or many user needs?	
11.	How difficult is conversion and installation?	
12.	How effective/automated are startup, backup, and recovery?	
13.	Was the application designed for multiple sites/organizations?	
14.	Was the application designed to facilitate change?	
	Value Adjustment Factor →	

1.3 Determine Function Points

Unadjusted Function Points x (0.65 + 0.01 x Value Adjustment Factor) →

Product metrics for Design

- These metrics place emphasis on the architectural structure and effectiveness of modules or components within the architecture
- They are “black box” in that they do not require any knowledge of the inner workings of a particular software component
- Card and Glass define three software design complexity measures.

Hierarchical Architecture Metrics

- **Fan out:** the number of modules immediately subordinate to the module *i*, that is, the number of modules directly invoked by module

- **Structural complexity**
  - $S(i) = f^2_{out}(i)$, where f_{out}(i) is the "fan out" of module

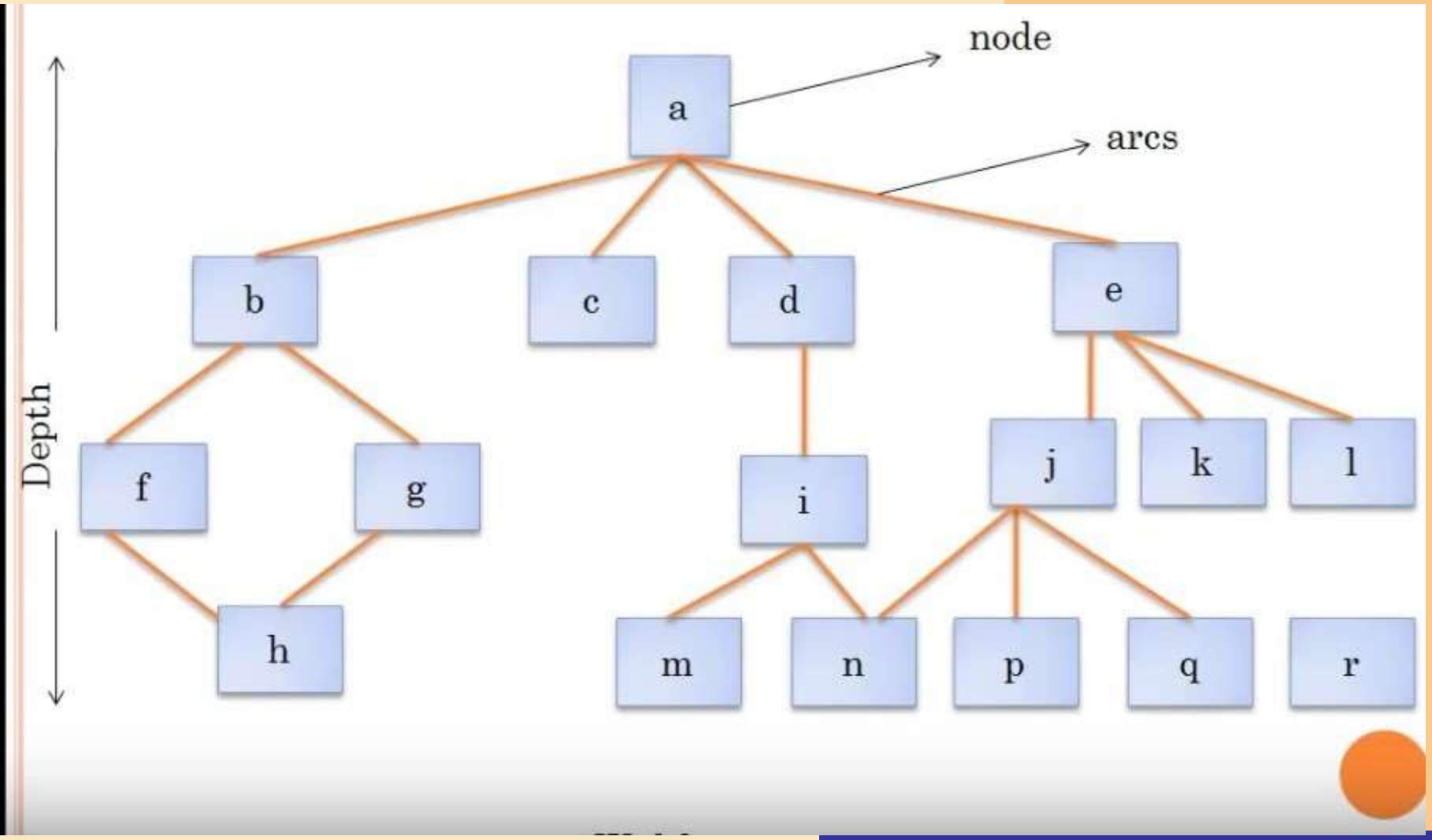
- **Data complexity**
  - $D(i) = v(i)/[f_{out}(i) + 1]$, where v(*i*) is the number of input and output variables that are passed to and from module

- **System complexity**
  - $C(i) = S(i) + D(i)$

- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to greater likelihood that the integration and testing effort will also increase

Hierarchical Architecture Metrics (continued)

- Shape complexity
  - `size = n + a`, where `n` is the number of nodes and `a` is the number of arcs
  - Allows different program software architectures to be compared in a straightforward manner
- Connectivity density (i.e., the arc-to-node ratio)
  - `r = a/n`
  - May provide a simple indication of the coupling in the software architecture


node
arcs
a
b
c
d
e
f
g
h
i
j
k
l
m
n
p
q
r
Depth

- Size= 17+18=35
- Depth=4, the longest path from the root (top) node to a leaf node
- Width=6, Maximum number of nodes at any one level of the architecture

- R=a/n
- R=18/17=1.06

Metrics for Design Model

- ✓ DSQI(Design Structure Quality Index)
- ✓ US air force has designed the DSQI
- ✓ Compute s1 to s7 from data and architectural design
- ✓ S1:Total number of modules
- ✓ S2:Number of modules whose correct function depends on the data input
- ✓ S3:Number of modules whose function depends on prior processing
- ✓ S4:Number of data base items
- ✓ S5:Number of unique database items
- ✓ S6: Number of database segments
- ✓ S7:Number of modules with single entry and exit
- ✓ Once values S1 through S7 are determined for a computer program, the following intermediate values can be computed

Metrics for Design Model

- ✓ Calculate D1 to D6 from s1 to s7 as follows:
- ✓ D1=1 if standard design is followed otherwise D1=0
- ✓ D2(module independence)=(1-(s2/s1))
- ✓ D3(module not depending on prior processing)=(1-(s3/s1))
- ✓ D4(Data base size)=(1-(s5/s4))
- ✓ D5(Database compartmentalization)=(1-(s6/s4)
- ✓ D6(Module entry/exit characteristics)=(1-(s7/s1))

Metrics for Design Model

- ✓ DSQI=sigma of WiDi
- ✓ i=1 to 6,Wi is weight assigned to Di
- ✓ If sigma of wi is 1 then all weights are equal to 0.167(if all Di are weighted equally , then wi=0.167)
- ✓ DSQI of present design be compared with past DSQI. If DSQI is significantly lower than the average, further design work and review are indicated

METRIC FOR SOURCE CODE

- HSS(Halstead Software science)
- Primitive measure that may be derived after the code is generated or estimated once design is complete

- n_1 = the number of distinct operators that appear in a program
- n_2 = the number of distinct operands that appear in a program
- N_1 = the total number of operator occurrences.
- N_2 = the total number of operand occurrence.

- Overall program length N can be computed:

- $N = n_1 \log2\ n_1 + n_2 \log2\ n_2$

- Program Volume may be defined

- $V = N \log_2 (n_1 + n_2)$

- Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume(number of bits required to specify a program),language level, and other features such as development effort, development time and even projected number of faults in the software.

Example

```
if (k < 2)
{
   if (k > 3)
      x = x*k;
}
```

- n_1 = the number of distinct operators that appear in a program
- n_2 = the number of distinct operands that appear in a program
- N_1 = the total number of operator occurrences.
- N_2 = the total number of operand occurrence

- Distinct operators: if ( ) { } > < = * ;
- Distinct operands: k 2 3 x
- $n_1 = 10$
- $n_2 = 4$
- $N_1 = 13$
- $N_2 = 7$

METRIC FOR TESTING

- Program Level and Effort
- PL = $1/[(n_1 / 2) \times (N_2 / n_2\ l)]$
- e = V/PL

METRICS FOR MAINTENANCE

- M_t = the number of modules in the current release
- F_c = the number of modules in the current release that have been changed
- F_a = the number of modules in the current release that have been added.
- F_d = the number of modules from the preceding release that were deleted in the current release

- The Software Maturity Index (SMI), is defined as:

- $SMI = [M_{t-(}F_c + F_{a+} F_{d)/} M_t]$
  - As the SMI (i.e., the fraction) approaches 1.0, the software product begins to stabilize
  - The average time to produce a release of a software product can be correlated with the SMI