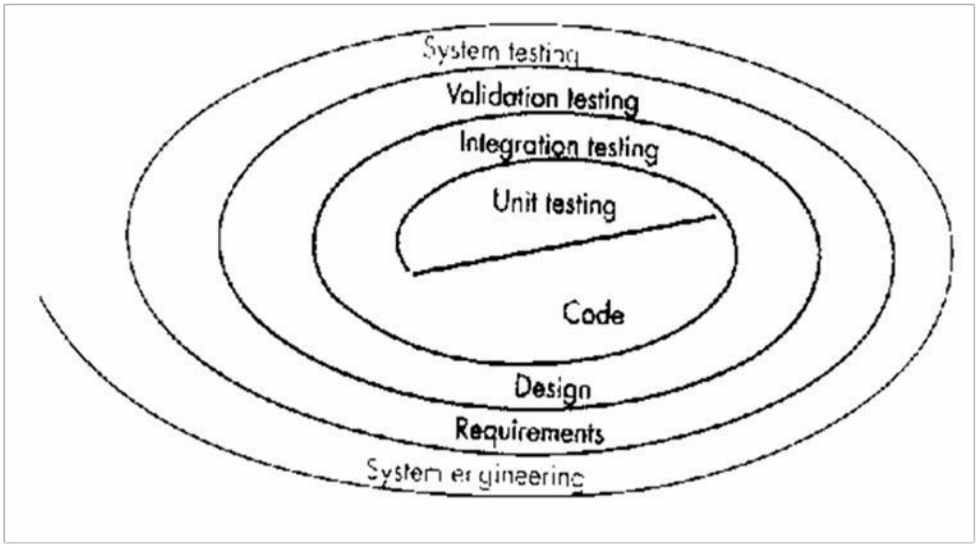
UNIT-IV

A strategic Approach for Software testing

- Software Testing
  - One of the important phases of software development
  - Testing is the process of execution of a program with the intention of finding errors
  - Involves 40% of total project cost
  - Testing Strategy
  - A road map that incorporates test planning, test case design, test execution and resultant data collection and execution
  - **Validation** refers to a different set of activities that ensures that the software is traceable to the customer requirements.
  - V&V encompasses a wide array of Software Quality Assurance
  - Perform Formal Technical reviews(FTR) to uncover errors during software development
  - Begin testing at component level and move outward to integration of entire component based system.
  - Adopt testing techniques relevant to stages of testing
  - Testing can be done by software developer and independent testing group
  - Testing and debugging are different activities. Debugging follows testing
  - Low level tests verifies small code segments.
  - High level tests validate major system functions against customer requirements

Testing Strategies for Conventional Software
1)Unit Testing
2)Integration Testing
3)Validation Testing and
4)System Testing

Spiral Representation for Conventional Software



Criteria for completion of software testing

- No body is absolutely certain that software will not fail
- Based on statistical modeling and software reliability models
- 95 percent confidence(probability) that 1000 CPU hours of failure free operation is at least 0.995

Software Testing

- Two major categories of software testing
- ❖ Black box testing
- ❖ White box testing

Black box testing

Treats the system as black box whose behavior can be determined by studying its input and related output
Not concerned with the internal structure of the program

Black Box Testing

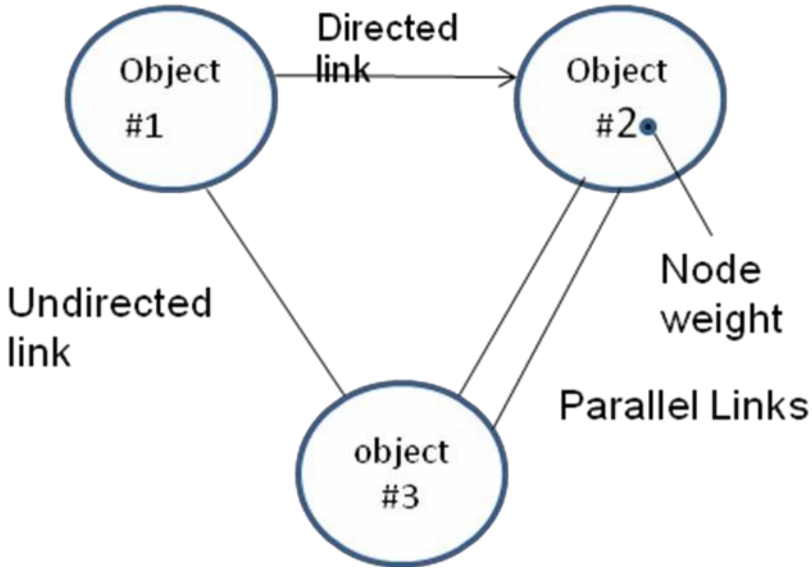
- It focuses on the functional requirements of the software ie it enables the sw engineer to derive a set of input conditions that fully exercise all the functional requirements for that program.
- Concerned with functionality and implementation

1)Graph based testing method

2)Equivalence partitioning

Graph based testing

- Draw a graph of objects and relations
- Devise test cases t uncover the graph such that each object and its relationship exercised.



Equivalence partitioning

- Divides all possible inputs into classes such that there are a finite equivalence classes.
- Equivalence class

-- Set of objects that can be linked by relationship

- Reduces the cost of testing
- Example
- Input consists of 1 to 10
- Then classes are n<1,1<=n<=10,n>10
- Choose one valid class with value within the allowed range and two invalid classes where values are greater than maximum value and smaller than minimum value.

Boundary Value analysis

- Select input from equivalence classes such that the input lies at the edge of the equivalence classes
- Set of data lies on the edge or boundary of a class of input data or generates the data that lies at the boundary of a class of output data

Example

- If 0.0<=x<=1.0
- Then test cases (0.0,1.0) for valid input and (-0.1 and 1.1) for invalid input

Orthogonal array Testing

- To problems in which input domain is relatively small but too large for exhaustive testing

Example

- Three inputs A,B,C each having three values will require 27 test cases
- L9 orthogonal testing will reduce the number of test case to 9 as shown below

A	B	C
1	1	1
1	2	2
1	3	3
2	1	3
2	2	3
2	3	1
3	1	3

3	2	1
3	3	2

White Box testing

- Also called glass box testing
- Involves knowing the internal working of a program
- Guarantees that all independent paths will be exercised at least once.
- Exercises all logical decisions on their true and false sides
- Executes all loops
- Exercises all data structures for their validity
- White box testing techniques

1. Basis path testing
2. Control structure testing

Basis path testing

- Proposed by Tom McCabe
- Defines a basic set of execution paths based on logical complexity of a procedural design
- Guarantees to execute every statement in the program at least once
- Steps of Basis Path Testing
- Draw the flow graph from flow chart of the program
- Calculate the cyclomatic complexity of the resultant flow graph
- Prepare test cases that will force execution of each path
- Three methods to compute Cyclomatic complexity number
- V(G)=E-N+2(E is number of edges, N is number of nodes
- V(G)=Number of regions
- V(G)= Number of predicates +1
- Control Structure testing
- Basis path testing is simple and effective
- It is not sufficient in itself
- Control structure broadens the basic test coverage and improves the quality of white box testing
- Condition Testing
- Data flow Testing
- Loop Testing
-

  Condition Testing
- --Exercise the logical conditions contained in a program module
- --Focuses on testing each condition in the program to ensure that it does contain errors
- --Simple condition
- E1<relation operator>E2
- --Compound condition
- simple condition<Boolean operator>simple condition

Data flow Testing

- Selects test paths according to the locations of definitions and use of variables in a program
- Aims to ensure that the definitions of variables and subsequent use is tested
- First construct a definition-use graph from the control flow of a program

Loop Testing

- Focuses on the validity of loop constructs
- Four categories can be defined

1. Simple loops
2. Nested loops
3. Concatenated loops
4. Unstructured loops

   Testing of simple loops

   -- N is the maximum number of allowable passes through the loop

Skip the loop entirely
Only one pass through the loop
Two passes through the loop
m passes through the loop where m>N
N-1,N,N+1 passes the loop

Nested Loops

1. Start at the innermost loop. Set all other loops to maximum values
2. Conduct simple loop test for the innermost loop while holding the outer loops at their minimum iteration parameter.
3. Work outward conducting tests for the next loop but keeping all other loops at minimum.

Concatenated loops

- Follow the approach defined for simple loops, if each of the loop is independent of other.
- If the loops are not independent, then follow the approach for the nested loops

Unstructured Loops

- Redesign the program to avoid unstructured loops

Validation Testing

- It succeeds when the software functions in a manner that can be reasonably expected by the customer.

1)Validation Test Criteria

2)Configuration Review

3)Alpha And Beta Testing

System Testing

- Its primary purpose is to test the complete software.

1)Recovery Testing

2)Security Testing

3Stress Testing and

4)Performance Testing

The Art of Debugging

- Debugging occurs as a consequences of successful testing.
- Debugging Stratergies

1)Brute Force Method.

2)Back Tracking

3)Cause Elimination and

4)Automated debugging

- Brute force

-- Most common and least efficient

-- Applied when all else fails

-- Memory dumps are taken

-- Tries to find the cause from the load of information

- Back tracking

-- Common debugging approach

-- Useful for small programs

-- Beginning at the system where the symptom has been uncovered, the source code traced backward until the site of the cause is found.

- Cause Elimination

-- Based on the concept of Binary partitioning

-- A list of all possible causes is developed and tests are conducted to eliminate each

Software Quality

- Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- 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)

- McCall’s quality factors

1. Product operation
    a. Correctness
    b. Reliability
    c. Efficiency
    d. Integrity
    e. Usability
2. Product Revision
    a. Maintainability
    b. Flexibility
    c. Testability

3. Product Transition
    a. Portability
    b. Reusability
    c. Interoperability

ISO 9126 Quality Factors

1.Functionality

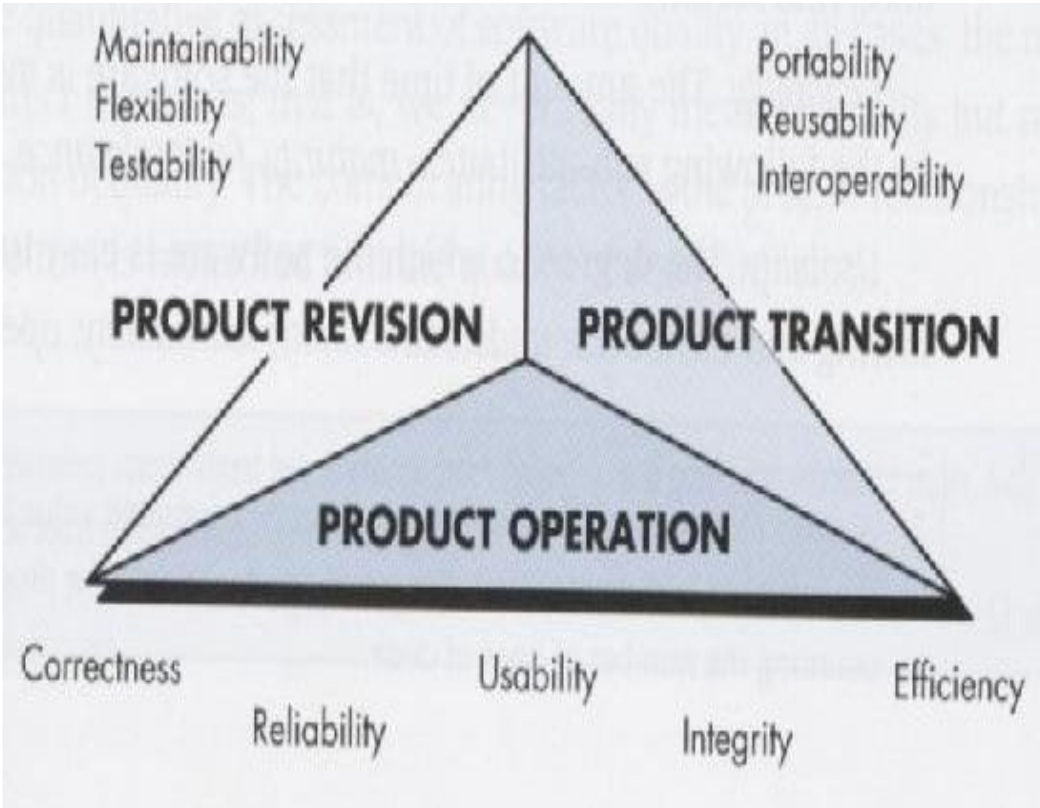
2.Reliability

3.Usability

4.Efficiency

5.Maintainability

6.Portability



Product metrics

- Product metrics for computer software helps us to assess quality.
- Measure

-- Provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process

- Metric(IEEE 93 definition)

-- A quantitative measure of the degree to which a system, component or process possess a given attribute

- Indicator

-- 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

- **<u>Product metrics for the Analysis model</u>**

❖ Function point Metric

- First proposed by Albrecht
- Measures the functionality delivered by the system
- FP computed from the following parameters

1) Number of external inputs(EIS)
2) Number external outputs(EOS)
3) Number of external Inquiries(EQS)
4) Number of Internal Logical Files(ILF)
5) Number of external interface files(EIFS)

Each parameter is classified as simple, average or complex and weights are assigned as follows

Information Domain	Count	Simple	avg	Complex
EIS		3	4	6
EOS		4	5	7
EQS		3	4	6
ILFS		7	10	15
EIFS		5	7	10

FP=Count total *[0.65+0.01*E(Fi)]

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
- 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))
- 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
- 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:
- $N = n_1 \log2 n_1 + n_2 \log2 n_2$
- $V = N \log_2 (n_1 + n_2)$

METRIC FOR TESTING

- 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.
- Program Level and Effort
- $PL = 1/[(n_1 / 2) \text{ x } (N_2 / n_2 l)]$
- 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:
- $SMI = [M_{t\,-}\,{}_{(}F_c + F_{a\,+}\,F_{d)/}\,M_t\ ]$

METRICS FOR PROCESS AND PROJECTS

1) SOFTWARE MEASUREMENT

Software measurement can be categorized in two ways.

(1) ***Direct measures*** of the software engineering process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.

(2) ***Indirect measures*** of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities"

1.1 Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the *size* of the software that has been produced.

To develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:

- Errors per KLOC (thousand lines of code).
- Defects per KLOC.
- $ per LOC.
- Page of documentation per KLOC.

In addition, other interesting metrics can be computed:

- Errors per person-month.
- LOC per person-month.
- $ per page of documentation.

1.2 Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht, who suggested a measure called the ***function point***. **Function points** are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

- ✓ **Proponents claim** that FP is programming language independent, making it ideal for application using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.
- ✓ **Opponents claim** that the method requires some “sleight of hand ” in that computation is basedsubjective rather than objective data, that counts of the information domain can be difficult to collect after the fact, and that FP has no direct physical meaning- it’s just a number.

Typical Function-Oriented Metrics:

- errors per FP (thousand lines of code)
- defects per FP
- $ per FP
- pages of documentation per FP
- FP per person-month

1.3) Reconciling Different Metrics Approaches

The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design.

Function points and LOC based metrics have been found to be relatively accurate predictors of software development effort and cost.

1.4) Object Oriented Metrics:

Conventional software project metrics (LOC or FP) can be used to estimate object oriented software projects. Lorenz and Kidd suggest the following set of metrics for OO projects:

- **Number of scenario scripts**: A scenario script is a detailed sequence of steps that describes the interaction between the user and the application.
- **Number of key classes**: Key classes are the “highly independent components that are defined early in object-oriented analysis.
- **Number of support classes**: Support classes are required to implement the system but are not immediately related to the problem domain.
- **Average number of support classes per key class**: Of the average number of support classes per key class were known for a given problem domain estimation would be much simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes.

- **Number of subsystems**: A subsystem is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in ehic work on subsystems is partitioned among project staff.

1.5) Use-Case Oriented Metrics

Use-cases describe user-visible functions and features that are basic requirements for a system. The use-cases is directly proportional to the size of the application in LOC and to the number of use-cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

Because use-cases can be created at vastly different levels of abstraction, there is no standard size for a use-case. Without a standard measure of what a use-case is, its application as a normalization measure is suspect.

1.6) Web Engineering Project Metrics

The objective of all web engineering projects is to build a Web application that delivers a combination of content and functionality to the end-user.

- Number of static Web pages: These pages represent low relative complexity and generally require less effort to construct than dynamic pages. This measures provides an indication of the overall size of the application and the effort required to develop it.
- Number of dynamic Web pages: : Web pages with dynamic content are essential in all e-commerce applications, search engines, financial application, and many other Web App categories. These pages represent higher relative complexity and require more effort to construct than static pages. This measure provides an indication of the overall size of the application and the effort required to develop it.
- Number of internal page link: Internal page links are pointers that provide an indication of the degree of architectural coupling within the Web App.
- Number of persistent data objects: As the number of persistent data objects grows, the complexity of the Web App also grows, and effort to implement it increases proportionally.
- Number of external systems interfaced: As the requirement for interfacing grows, system complexity and development effort also increase.
- Number of static content objects: Static content objects encompass static text- based, graphical, video, animation, and audio information that are incorporated within the Web App.
- Number of dynamic content objects: Dynamic content objects are generated based on end-user actions and encompass internally generated text-based, graphical, video, animation, and audio information that are incorporated within the Web App.
- Number of executable functions: An executable function provides some computational service to the end-user. As the number of executable functions increases, modeling and construction effort also increase.

2) METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process.

2.1 Measuring Quality

The measures of software quality are correctness, maintainability, integrity, and usability. These measures will provide useful indicators for the project team.

- **Correctness.** Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- **Maintainability.** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple time-oriented metric is ***mean-time-tochange*** (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.
- **Integrity.** Attacks can be made on all three components of software: programs, data, and documents.

To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

$$\text{integrity} = \sum [1 - (\text{threat} \times (1 - \text{security}))]$$

- **Usability:** Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

2.2 Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner:

$$DRE = E/(E + D)$$

where E is the number of errors found before delivery of the software to the end-user and
D is the number of defects found after delivery.

Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

$DRE_i = E_i/(E_i+E_{i+1})$

E_i is the number of errors found during software engineering activity i and
E_{i+1} is the number of errors found during software engineering activity $i+1$ that are traceable to errors that were not discovered in software engineering activity i.

A quality objective for a software team (or an individual software engineer) is to achieve DRE that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.