MALLA REDDY UNIVERSITY

LECTURE NOTES

On

MR22-1CS0146: OBJECT ORIENTED SOFTWARE ENGINEERING

II Year B.Tech

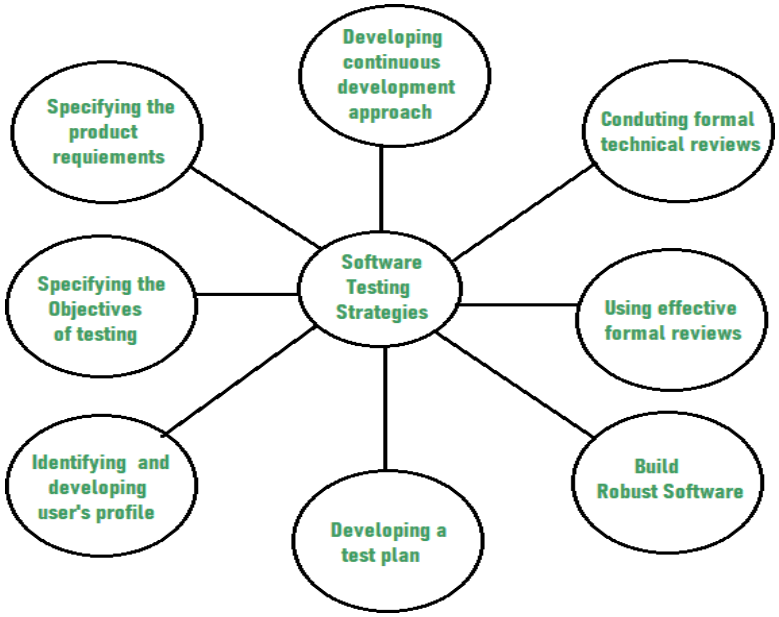
UNIT-IV

Testing Strategies: A strategic approach to software testing, test strategies for conventional software, black-box and white-box testing, validation testing, system testing, the art of debugging.
Product metrics: Software quality, metrics for analysis model, metrics for design model, metrics for source code, metrics for testing, metrics for maintenance.

A strategic Approach for Software testing:

Software Testing is a type of investigation to find out if there is any default or error present in the software so that the errors can be reduced or removed to increase the quality of the software and to check whether it fulfills the specifies requirements or not.

The main objective of software testing is to design the tests in such a way that it systematically finds different types of errors without taking much time and effort so that less time is required for the development of the software. The overall strategy for testing software includes:



1. **Before testing starts, it's necessary to identify and specify the requirements of the product in a quantifiable manner.**
   Different characteristics quality of the software is there such as maintainability that means the ability to update and modify, the probability that means to find and estimate any risk, and usability that means how it can easily be used by the customers or end-users. All these characteristic qualities should be specified in a particular order to obtain clear test results without any error.

2. **Specifying the objectives of testing in a clear and detailed manner.**
   Several objectives of testing are there such as effectiveness that means how effectively the software can achieve the target, any failure that means inability to fulfill the requirements and perform functions, and the cost of defects or errors that mean the cost required to fix the error. All these objectives should be clearly mentioned in the test plan.

3. **For the software, identifying the user's category and developing a profile for each user.**
   Use cases describe the interactions and communication among different classes of users and the system to achieve the target. So as to identify the actual requirement of the users and then testing the actual use of the product.

4. **Developing a test plan to give value and focus on rapid-cycle testing.**
   Rapid Cycle Testing is a type of test that improves quality by identifying and measuring the any changes that need to be required for improving the process of software. Therefore, a test plan is an important and effective document that helps the tester to perform rapid cycle testing.

5. **Robust software is developed that is designed to test itself.**
   The software should be capable of detecting or identifying different classes of errors. Moreover, software design should allow automated and regression testing which tests the software to find out if there is any adverse or side effect on the features of software due to any change in code or program.

6. **Before testing, using effective formal reviews as a filter.**
   Formal technical reviews is technique to identify the errors that are not discovered yet. The effective technical reviews conducted before testing reduces a significant amount of testing efforts and time duration required for testing software so that the overall development time of software is reduced.

7. **Conduct formal technical reviews to evaluate the nature, quality or ability of the test strategy and test cases.**
   The formal technical review helps in detecting any unfilled gap in the testing approach. Hence, it is necessary to evaluate the ability and quality of the test strategy and test cases by technical reviewers to improve the quality of software.

8. **For the testing process, developing a approach for the continuous development.**
   As a part of a statistical process control approach, a test strategy that is already measured should be used for software testing to measure and control the quality during the development of software.

Testing Strategies for Conventional(o-o) Software

- There are many strategies that can be used to test software.
- At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
    - This approach simply does not work. It will result in buggy software.
- At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.
    - This approach, although less appealing to many, can be very effective.

Types:

1) Unit Testing
2) Integration Testing
3) Validation Testing and
4) System Testing

1) Unit Testing:

Unit testing is a type of software testing where individual units or components of a software are tested. It is concerned with functional correctness of the standalone modules. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

Why Unit Testing?

Unit Testing is important because software developers sometimes try saving time doing minimal unit testing and this is myth because inappropriate unit testing leads to high cost Defect fixing during System Testing, Integration Testing and even Beta Testing after application is built. If proper unit testing is done in early development, then it saves time and money in the end.

Here, are the key reasons to perform unit testing:

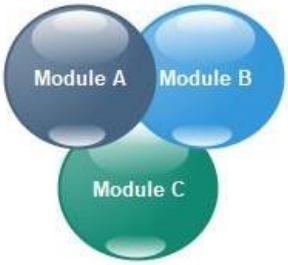
1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the code base and enables them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code until the tests run again.

2) Integration Testing

Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group. The focus of the integration testing level is to expose defects at the time of interaction between integrated components or units.

Unit testing uses modules for testing purpose, and these modules are combined and tested in integration testing. The Software is developed with a number of software modules that are coded by different coders or programmers. The goal of integration testing is to check the correctness of communication among all the modules.

Tested in Unit Testing



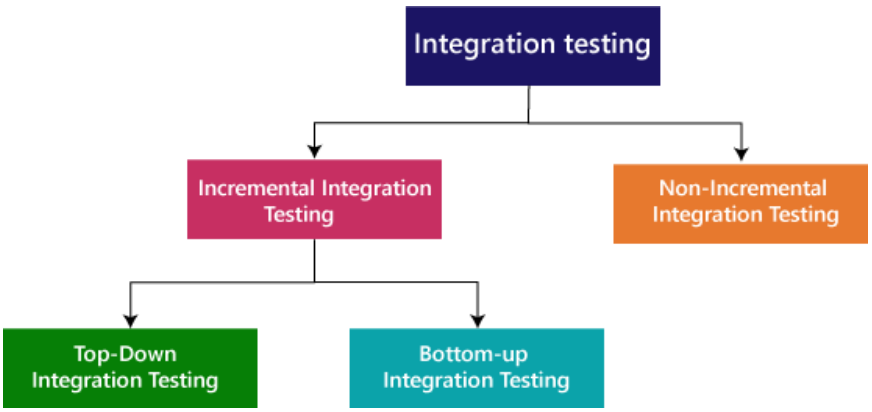
Under Integration Testing

Once all the components or modules are working independently, then we need to check the data flow between the dependent modules is known as **integration testing**.

Types of Integration Testing

Integration testing can be classified into two parts:

- **Incremental integration testing**
- **Non-incremental integration testing**



Incremental Approach

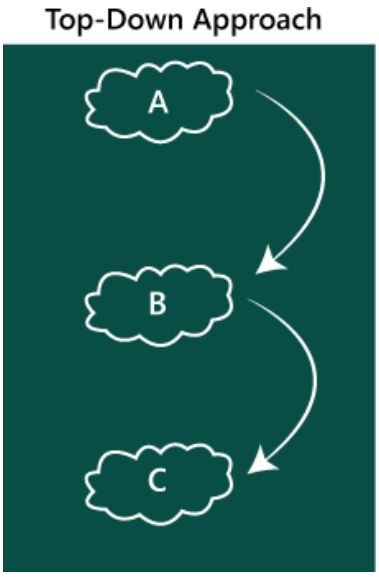
In the Incremental Approach, modules are added in ascending order one by one or according to need. The selected modules must be logically related. Generally, two or more than two modules are added and tested to determine the correctness of functions. The process continues until the successful testing of all the modules.

Incremental integration testing is carried out by further methods:

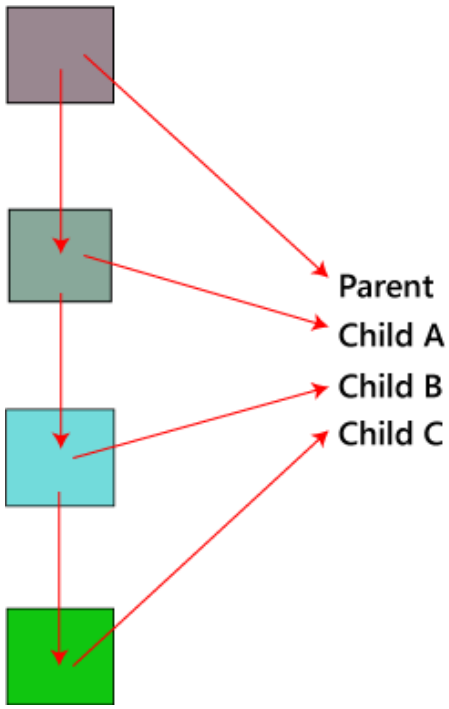
- Top-Down approach
- Bottom-Up approach

Top-Down Approach

The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules. Major design flaws can be detected and fixed early because critical modules tested first. In this type of method, we will add the modules incrementally or one by one and check the data flow in the same order.



In the top-down approach, we will be ensuring that the module we are adding is the **child of the previous one like Child C is a child of Child B** and so on as we can see in the below image:



Advantages:

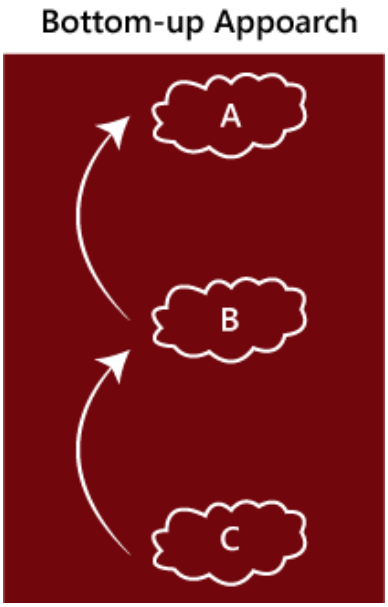
- Identification of defect is difficult.
- An early prototype is possible.

Disadvantages:

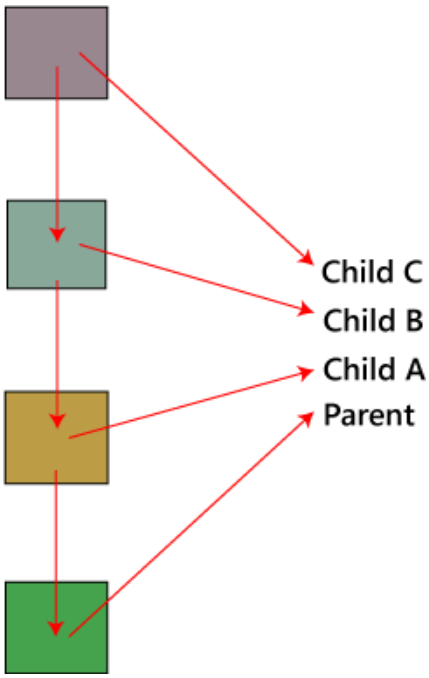
- Due to the high number of stubs, it gets quite complicated.
- Lower level modules are tested inadequately.
- Critical Modules are tested first so that fewer chances of defects.

Bottom-Up Method

The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules. Top level critical modules are tested at last, so it may cause a defect. Or we can say that we will be adding the modules from **bottom to the top** and check the data flow in the same order.



In the bottom-up method, we will ensure that the modules we are adding **are the parent of the previous one** as we can see in the below image:



Advantages

- Identification of defect is easy.
- Do not need to wait for the development of all the modules as it saves time.

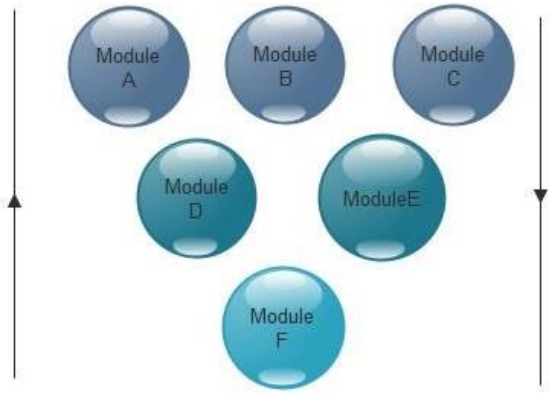
Disadvantages

- Critical modules are tested last due to which the defects can occur.
- There is no possibility of an early prototype.

In this, we have one addition approach which is known as **hybrid testing**.

Hybrid Testing Method

In this approach, both **Top-Down** and **Bottom-Up** approaches are combined for testing. In this process, top-level modules are tested with lower level modules and lower level modules tested with high-level modules simultaneously. There is less possibility of occurrence of defect because each module interface is tested.



Hybrid Method

Advantages

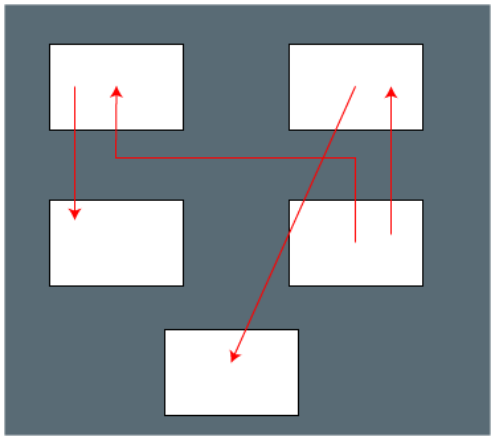
- The hybrid method provides features of both Bottom Up and Top Down methods.
- It is most time reducing method.
- It provides complete testing of all modules.

Disadvantages

- This method needs a higher level of concentration as the process carried out in both directions simultaneously.
- Complicated method.

Non- incremental integration testing

We will go for this method, when the data flow is very complex and when it is difficult to find who is a parent and who is a child. And in such case, we will create the data in any module bang on all other existing modules and check if the data is present. Hence, it is also known as the **Big bang method**.



3. Validation Testing

Verification and Validation Testing

Verification testing

Verification testing includes different activities such as business requirements, system requirements, design review, and code walkthrough while developing a product.

It is also known as static testing, where we are ensuring that "**we are developing the right product or not**". And it also checks that the developed application fulfilling all the requirements given by the client.

Validation testing

Validation testing is testing where tester performed functional and non-functional testing. Here **functional testing** includes Unit Testing (UT), Integration Testing (IT) and System Testing (ST), and **non-functional** testing includes User acceptance testing (UAT).

Validation testing is also known as dynamic testing, where we are ensuring that **"we have developed the product right."** And it also checks that the software meets the business needs of the client.

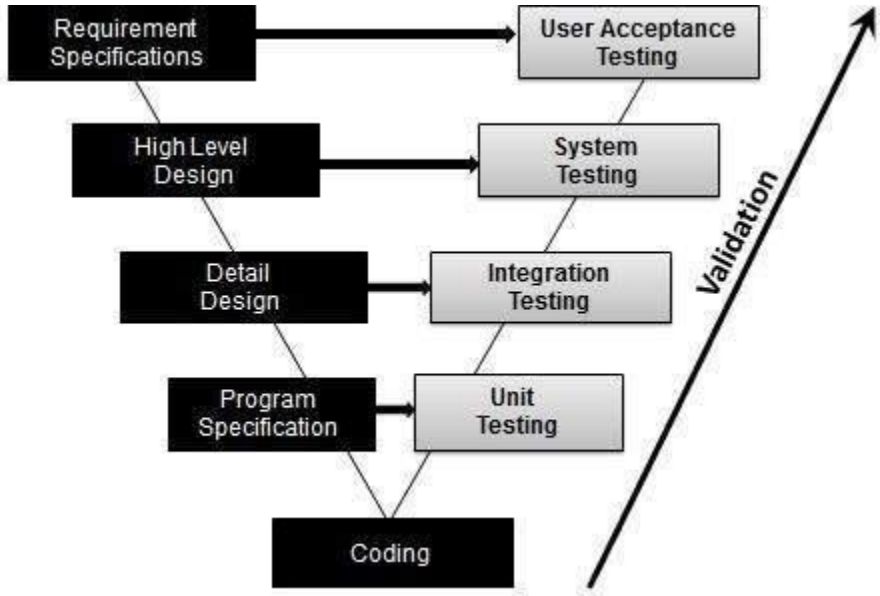
The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

It answers to the question, Are we building the right product?

Validation Testing - Workflow:

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.



4. System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system). The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware. System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.

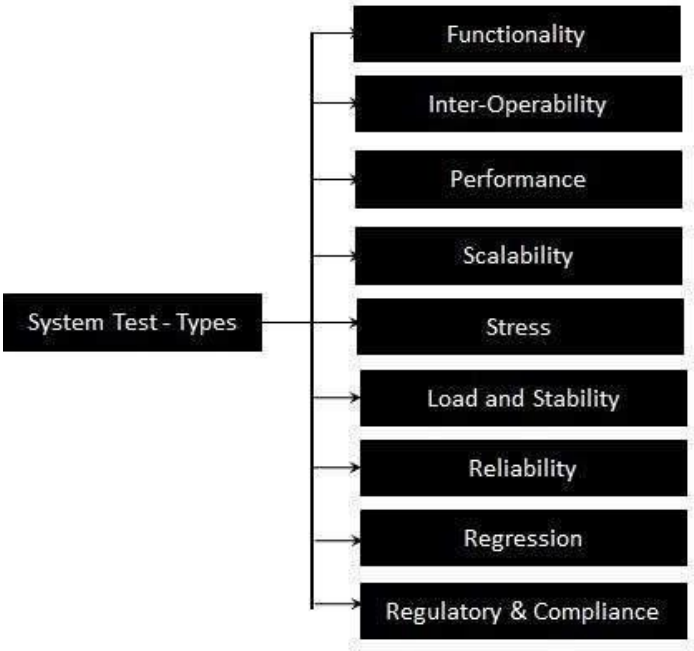
To check the end-to-end flow of an application or the software as a user is known as **System testing**. In this, we navigate (go through) all the necessary modules of an application and check if the end features or the end business works fine, and test the product as a whole system.

It is **end-to-end testing** where the testing environment is similar to the production environment.

System Testing includes the following steps.

- Verification of input functions of the application to test whether it is producing the expected output or not.
- Testing of integrated software by including external peripherals to check the interaction of various components with each other.
- Testing of the whole system for End to End testing.
- Behavior testing of the application via a user's experience

Types of System Tests:



Software Testing

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

Black box testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

How to do Black Box Testing?

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones -

- **Functional testing** - This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** - This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Partitioning:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Analysis:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Equivalence Partitioning Testing

Equivalence Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc. also called as equivalence class partitioning. It is abbreviated as ECP. It is a software testing technique that divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.

An advantage of this approach is it reduces the time required for performing testing of a software due to less number of test cases.

Example:

The Below example best describes the equivalence class Partitioning:

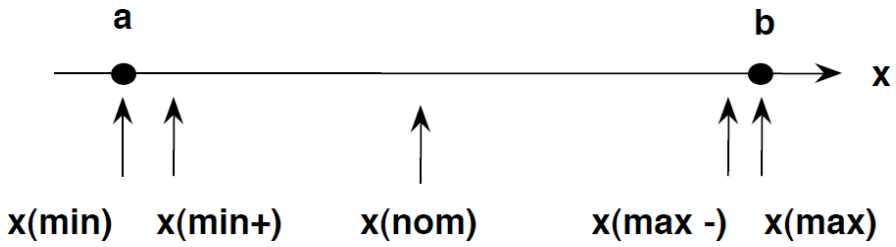
Assume that the application accepts an integer in the range 100 to 999
Valid Equivalence Class partition: 100 to 999 inclusive.
Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters.

Boundary Value Analysis

Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".
- The basic idea in boundary value testing is to select input variable values at their:

1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum



Example: Input Box should accept the Number 1 to 10

Here we will see the Boundary Value Test Cases

Test Scenario Description	Expected Outcome
Boundary Value = 0	System should NOT accept
Boundary Value = 1	System should accept
Boundary Value = 2	System should accept

Boundary Value = 9	System should accept
Boundary Value = 10	System should accept
Boundary Value = 11	System should NOT accept

Example 1: Equivalence and Boundary Value

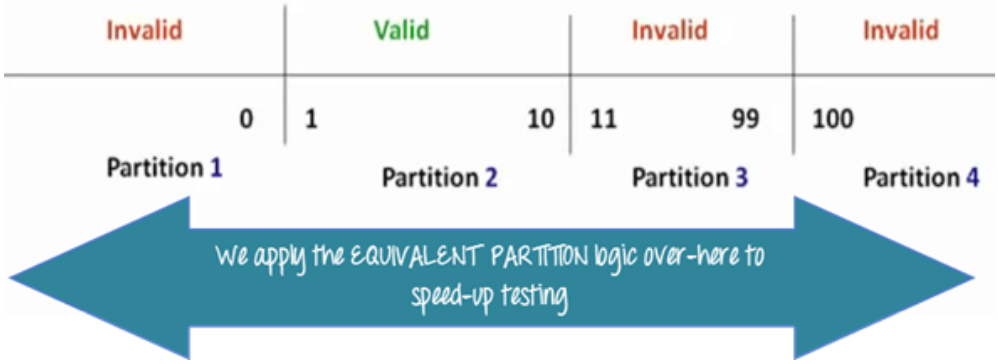
- Let's consider the behavior of Order Pizza Text Box Below
- Pizza values 1 to 10 is considered valid. A success message is shown.
- While value 11 to 99 are considered invalid for order and an error message will appear, **"Only 10 Pizza can be ordered"**

Order Pizza: [ ] Submit

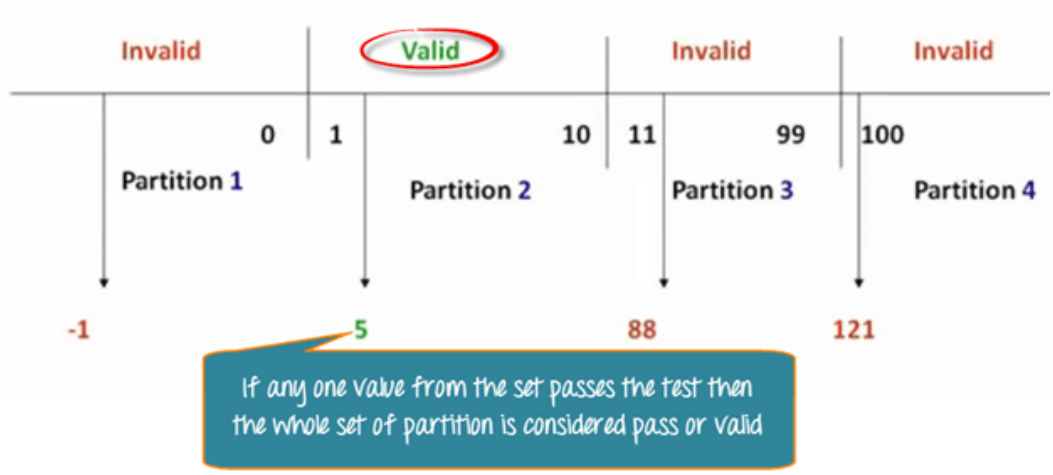
Here is the test condition

1. Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.

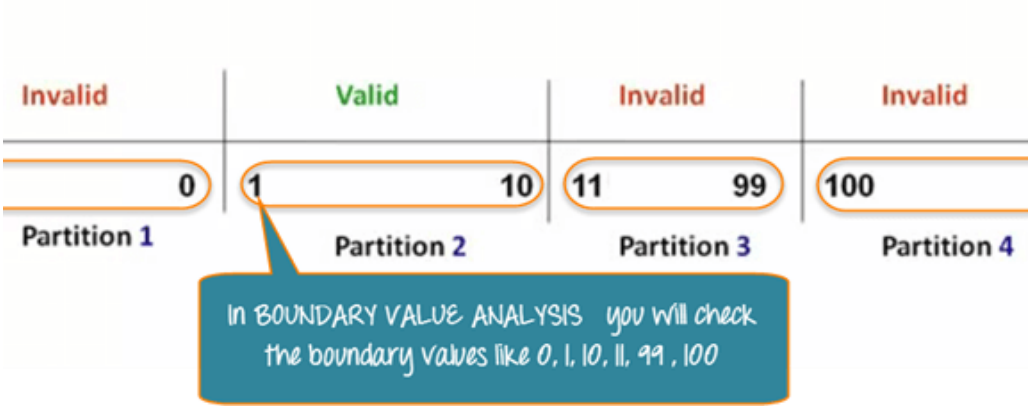
We cannot test all the possible values because if done, the number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same.



The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is **that if one condition/value in a partition passes all others will also pass**. Likewise**, if one condition in a partition fails, all other conditions in that partition will fail**.



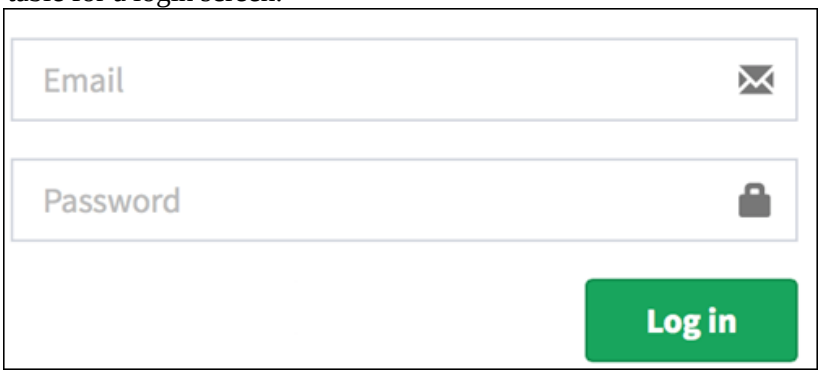
Boundary Value Analysis- in Boundary Value Analysis, you test boundaries between equivalence partitions



In our earlier example instead of checking, one value for each partition you will check the values at the partitions like 0, 1, 10, 11 and so on. As you may observe, you test values at **both valid and invalid boundaries**. Boundary Value Analysis is also called **range checking**.
Equivalence partitioning and boundary value analysis(BVA) are closely related and can be used together at all levels of testing.

Decision Table

A **Decision Table** is a tabular representation of inputs versus rules/cases/test conditions. It is a very effective tool used for both complex software testing and requirements management. Decision table helps to check all possible combinations of conditions for testing and testers can also identify missed conditions easily. The conditions are indicated as True(T) and False(F) values.
Example 1: How to make Decision Base Table for Login Screen
Let's create a decision table for a login screen.



The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

Legend:

- T – Correct username password
- F – Wrong username/password
- E – Error message is displayed
- H – Home screen is displayed

Interpretation:

- Case 1 – Username and password both were wrong. The user is shown an error message.
- Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
- Case 3 – Username was wrong, but the password was correct. The user is shown an error message.
- Case 4 – Username and password both were correct, and the user navigated to homepage

While converting this to test case, we can create 2 scenarios,

- Enter correct username and correct password and click on login, and the expected result will be the user should be navigated to homepage

And one from the below scenario

- Enter wrong username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter correct username and wrong password and click on login, and the expected result will be the user should get an error message
- Enter wrong username and correct password and click on login, and the expected result will be the user should get an error message

White Box Testing:

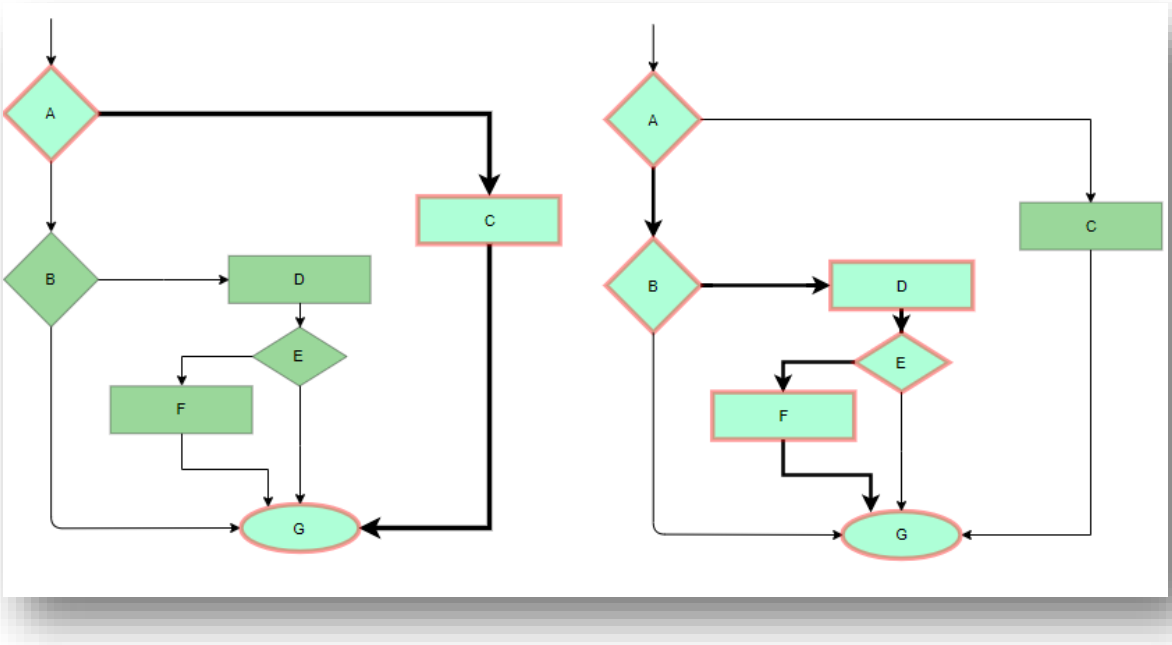
White box testing is a testing technique that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

White Box Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered.

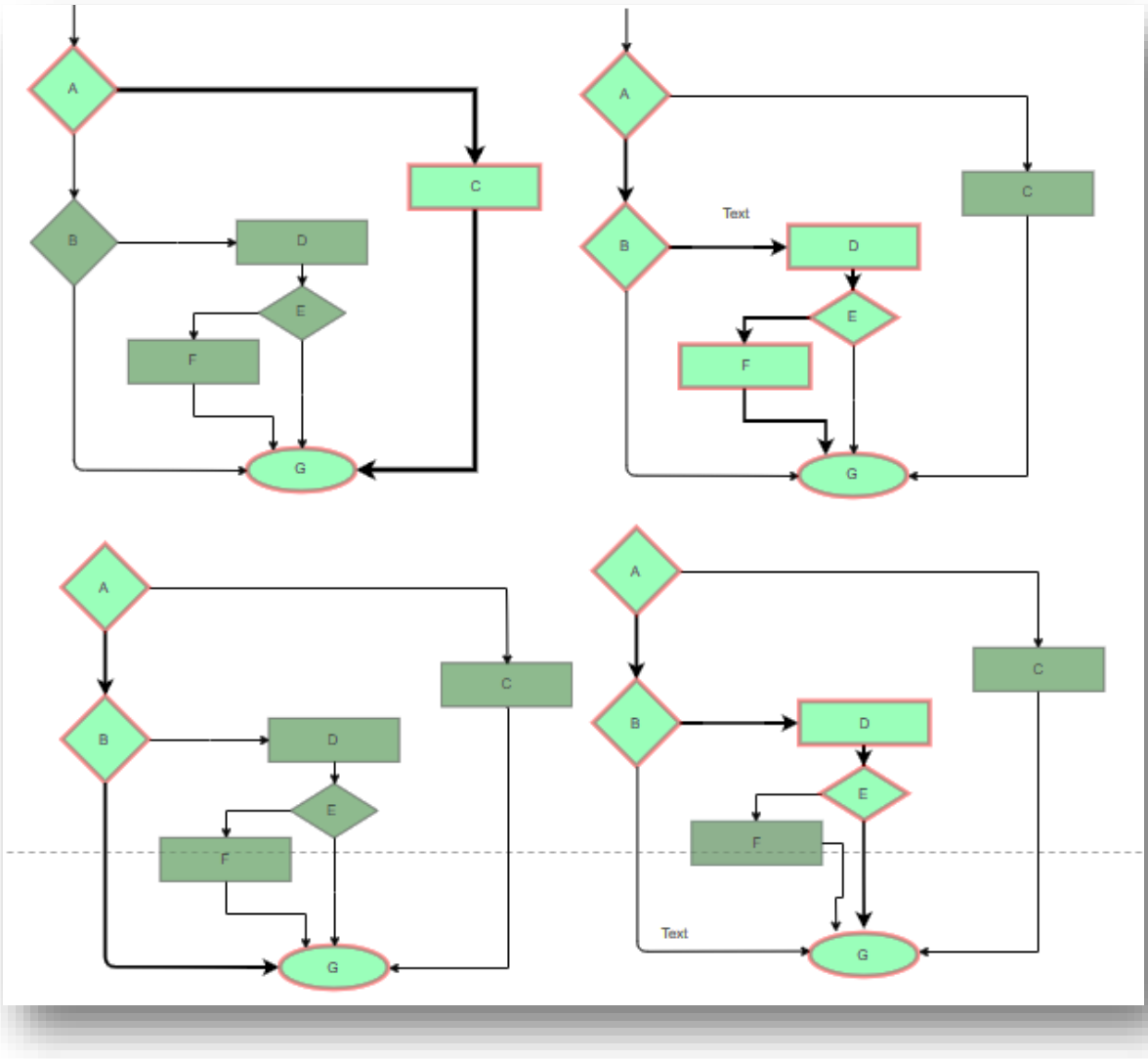
Statement coverage:

In this technique, the aim is to traverse all statement at least once. Hence, each line of code is tested. In case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



Statement Coverage Example

Branch Coverage: In this technique, test cases are designed so that each branch from all decision points are traversed at least once. In a flowchart, all edges must be traversed at least once.



4 test cases required such that all branches of all decisions are covered, i.e, all edges of flowchart are covered

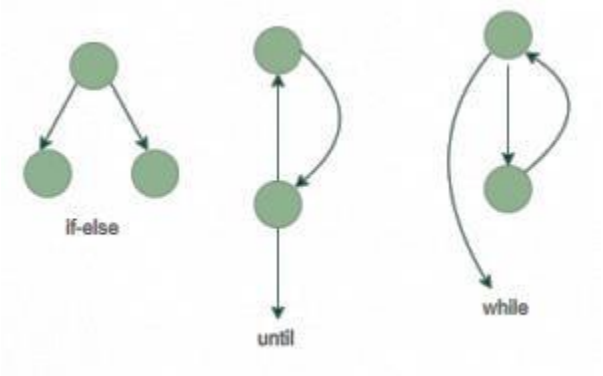
Basis Path Testing: In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.

Steps:

1. Make the corresponding control flow graph
2. Calculate the cyclomatic complexity
3. Find the independent paths
4. Design test cases corresponding to each independent path

Flow graph notation: It is a directed graph consisting of nodes and edges. Each node represents a sequence of statements, or a decision point. A predicate node is the one that represents a decision point that

contains a condition after which the graph splits. Regions are bounded by nodes and edges.



Cyclomatic Complexity: It is a measure of the logical complexity of the software and is used to define the number of independent paths. For a graph G, V(G) is its cyclomatic complexity.

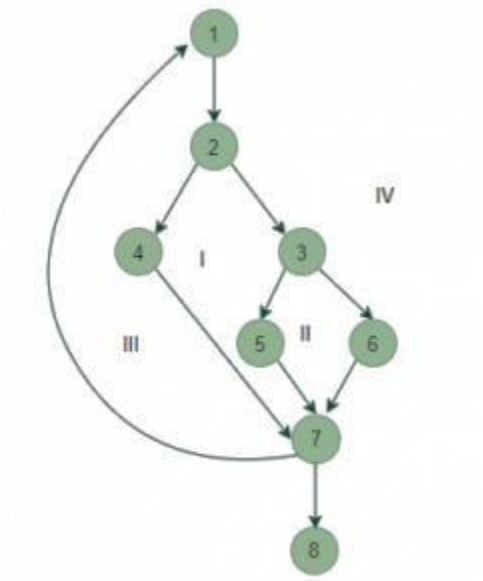
Calculating V(G):

V(G) = P + 1, where P is the number of predicate nodes in the flow graph

V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes

V(G) = Number of non-overlapping regions in the graph

Example:



V(G) = 4 (Using any of the above formulae)

No of independent paths = 4

- #P1: 1 – 2 – 4 – 7 – 8
- #P2: 1 – 2 – 3 – 5 – 7 – 8
- #P3: 1 – 2 – 3 – 6 – 7 – 8
- #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

Loop Testing: Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.

Simple loops: For simple loops of size n, test cases are designed that:

- Skip the loop entirely
- Only one pass through the loop
- 2 passes
- m passes, where m < n
- n-1 ans n+1 passes

Nested loops: For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.

Concatenated loops: Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

Software Quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

The modern view of a quality associated with a software product several quality methods such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show up, new tasks can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Product Metrics

In software development process, a working product is developed at the end of each successful phase. Each product can be measured at any stage of its development. Metrics are developed for these products so that they can indicate whether a product is developed according to the user requirements. If a product does not meet user requirements, then the necessary actions are taken in the respective phase.

Product metrics help software engineer to detect and correct potential problems before they result in catastrophic defects. In addition, product metrics assess the internal product attributes in order to know the efficiency of the following.

- Analysis, design, and code model
- Potency of test cases
- Overall quality of the software under development.

Various metrics formulated for products in the development process are listed below.

- **Metrics for analysis model:** These address various aspects of the analysis model such as system functionality, system size, and so on.
- **Metrics for design model:** These allow software engineers to assess the quality of design and include architectural design metrics, component-level design metrics, and so on.
- **Metrics for source code:** These assess source code complexity, maintainability, and other characteristics.
- **Metrics for testing:** These help to design efficient and effective test cases and also evaluate the effectiveness of testing.
- **Metrics for maintenance:** These assess the stability of the software product.

Metrics for the Analysis Model

There are only a few metrics that have been proposed for the analysis model. However, it is possible to use metrics for project estimation in the context of the analysis model. These metrics are used to examine the analysis model with the objective of predicting the size of the resultant system. Size acts as an indicator of increased coding, integration, and testing effort; sometimes it also acts as an indicator of complexity involved in the software design. Function point and lines of code are the commonly used methods for size estimation.

Function Point (FP) Metric

The function point metric, which was proposed by A.J Albrecht, is used to measure the functionality delivered by the system, estimate the effort, predict the number of errors, and estimate the number of components in the system.

Function point is derived by using a relationship between the complexity of software and the information domain value. Information domain values used in function point include the number of external inputs, external outputs, external inquires, internal logical files, and the number of external interface files.

Lines of Code (LOC)

Lines of code (LOC) is one of the most widely used methods for size estimation. LOC can be defined as the number of delivered lines of code, excluding comments and blank lines. It is highly dependent on the programming language used as code writing varies from one programming language to another. Fur example, lines of code written (for a large program) in assembly language are more than lines of code written in C++. From LOC, simple size-oriented metrics can be derived such as errors per KLOC (thousand lines of code), defects per KLOC, cost per KLOC, and so on. LOC has also been used to predict program complexity, development effort, programmer performance, and so on. For example, Hasltead proposed a number of metrics, which are used to calculate program length, program volume, program difficulty, and development effort.

Metrics for Specification Quality

To evaluate the quality of analysis model and requirements specification, a set of characteristics has been proposed. These characteristics include specificity, completeness, correctness, understandability, verifiability, internal and external consistency, &achievability, concision, traceability, modifiability, precision, and reusability. Most of the characteristics listed above are qualitative in nature. However, each of these characteristics can be represented by using one or more metrics. For example, if there are n_r requirements in a specification, then n_r can be calculated by the following equation.

$$n_r = n_f + n_{rf}$$

Where

n_f = number of functional requirements

n_{nf} = number of non-functional requirements.

In order to determine the specificity of requirements, a metric based on the consistency of the reviewer's understanding of each requirement has been proposed. This metric is represented by the following equation.

$$Q_1 = n_{ui}/n_r$$

Where

n_{ui} = number of requirements for which reviewers have same understanding

Q_1 = specificity.

Ambiguity of the specification depends on the value of Q. If the value of Q is close to 1 then the probability of having any ambiguity is less.

Completeness of the functional requirements can be calculated by the following equation.

$$Q_2 = n_u / [n_j * n_s]$$

Where

n_u = number of unique function requirements

n_i = number of inputs defined by the specification

n_s = number of specified state.

Q_2 in the above equation considers only functional requirements and ignores non-functional requirements. In order to consider non-functional requirements, it is necessary to consider the degree to which requirements have been validated. This can be represented by the following equation.

$$Q_3 = n_c / [n_c + n_{nv}]$$

Where

n_c= number of requirements validated as correct

n_{nv}= number of requirements, which are yet to be validated.

Metrics for Software Design

The success of a software project depends largely on the quality and effectiveness of the software design. Hence, it is important to develop software metrics from which meaningful indicators can be derived. With the help of these indicators, necessary steps are taken to design the software according to the user requirements. Various design metrics such as architectural design metrics, component-level design metrics, user-interface design metrics, and metrics for object-oriented design are used to indicate the complexity, quality, and so on of the software design.

Architectural Design Metrics

These metrics focus on the features of the program architecture with stress on architectural structure and effectiveness of components (or modules) within the architecture. In architectural design metrics, three software design complexity measures are defined, namely, structural complexity, data complexity, and system complexity. In hierarchical architectures (call and return architecture), say module 'j', structural complexity is calculated by the following equation.

$S(j) = f^2_{out}(j)$

Where

$f_{out}(j)$ = fan-out of module 'j' [Here, fan-out means number of modules that are subordinating module j].

Complexity in the internal interface for a module 'j' is indicated with the help of data complexity, which is calculated by the following equation.

$D(j) = V(j) / [f_{out}(j)+l]$

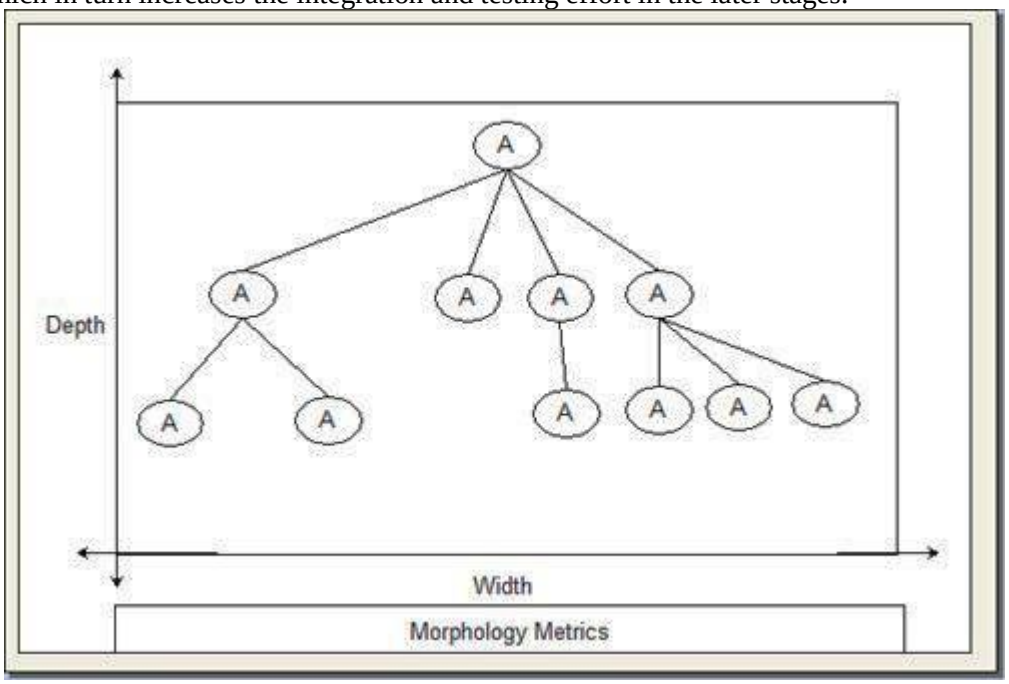
Where

V(j) = number of input and output variables passed to and from module 'j'.

System complexity is the sum of structural complexity and data complexity and is calculated by the following equation.

$C(j) = S(j) + D(j)$

The complexity of a system increases with increase in structural complexity, data complexity, and system complexity, which in turn increases the integration and testing effort in the later stages.



In addition, various other metrics like simple morphology metrics are also used. These metrics allow comparison of different program architecture using a set of straightforward dimensions. A metric can be developed by referring to call and return architecture. This metric can be defined by the following equation.

$$Size = n+a$$

Where

n = number of nodes

a= number of arcs.

For example, there are 11 nodes and 10 arcs. Here, Size can be calculated by the following equation.

$$Size = n+a = 11+10+21.$$

Depth is defined as the longest path from the top node (root) to the leaf node and width is defined as the maximum number of nodes at any one level.

Coupling of the architecture is indicated by arc-to-node ratio. This ratio also measures the connectivity density of the architecture and is calculated by the following equation.

$$r=a/n$$

Quality of software design also plays an important role in determining the overall quality of the software. Many software quality indicators that are based on measurable design characteristics of a computer program have been

proposed. One of them is Design Structural Quality Index (DSQI), which is derived from the information obtained from data and architectural design. To calculate DSQI, a number of steps are followed, which are listed below.

1. To calculate DSQI, the following values must be determined.
   - Number of components in program architecture (S_1)
   - Number of components whose correct function is determined by the Source of input data (S_2)
   - Number of components whose correct function· depends on previous processing (S_3)
   - Number of database items (S_4)
   - Number of different database items (S_5)
   - Number of database segments (S_6)
   - Number of components having single entry and exit (S_7).

2. Once all the values from S_1 to S_7 are known, some intermediate values are calculated, which are listed below.

Program structure (D_1): If discrete methods are used for developing architectural design then D_1= 1, else D_1 = 0

Module independence (D_2): $D_2 = 1-(S_2/S_1)$

Modules not dependent on prior processing (D_3): $D_3 = 1-(S_3/S_1)$

Database size (D_4): $D_4 = 1-(S_5/S_4)$

Database compartmentalization (D_5): $D_5 = 1-(S_6/S_4)$

Module entrance/exit characteristic (D_6): $D_6 = 1-(S_7/S_1)$

3. Once all the intermediate values are calculated, OSQI is calculated by the following equation.

$$DSQI = \sum W_i D_i$$

Where

i = 1 to 6

$\sum W_i = 1$ (W_i is the weighting of the importance of intermediate values).

In conventional software, the focus of component – level design metrics is on the internal characteristics of the software components; The software engineer can judge the quality of the component-level design by measuring module cohesion, coupling and complexity; Component-level design metrics are applied after procedural design is final. Various metrics developed for component-level design are listed below.

- **Cohesion metrics:** Cohesiveness of a module can be indicated by the definitions of the following five concepts and measures.
- **Data slice:** Defined as a backward walk through a module, which looks for values of data that affect the state of the module as the walk starts
- **Data tokens:** Defined as a set of variables defined for a module
- **Glue tokens:** Defined as a set of data tokens, which lies on one or more data slice
- **Superglue tokens:** Defined as tokens, which are present in every data slice in the module
- **Stickiness:** Defined as the stickiness of the glue token, which depends on the number of data slices that it binds.
- **Coupling Metrics:** This metric indicates the degree to which a module is connected to other modules, global data and the outside environment. A metric for module coupling has been proposed, which includes data and control flow coupling, global coupling, and environmental coupling.
  - Measures defined for data and control flow coupling are listed below.

d_i = total number of input data parameters

c_i = total number of input control parameters

d_o= total number of output data parameters

c_o= total number of output control parameters

\$1§ Measures defined for global coupling are listed below.

g_d= number of global variables utilized as data

g_c = number of global variables utilized as control

\$1§ Measures defined for environmental coupling are listed below.

w = number of modules called

r = number of modules calling the modules under consideration

By using the above mentioned measures, module-coupling indicator (m_c) is calculated by using the following equation.

$$m_c = K/M$$

Where

K = proportionality constant

$$M = d_i + (a*c_i) + d_o + (b*c_o) + g_d + (c*g_c) + w + r.$$

Note that K, a, b, and c are empirically derived. The values of m_c and overall module coupling are inversely proportional to each other. In other words, as the value of m_c increases, the overall module coupling decreases.

Complexity Metrics: Different types of software metrics can be calculated to ascertain the complexity of program control flow. One of the most widely used complexity metrics for ascertaining the complexity of the program is cyclomatic complexity.

Many metrics have been proposed for user interface design. However, layout appropriateness metric and cohesion metric for user interface design are the commonly used metrics. Layout Appropriateness (LA) metric is an important metric for user interface design. A typical Graphical User Interface (GUI) uses many layout entities such as icons, text, menus, windows, and so on. These layout entities help the users in completing their tasks easily. In to complete a given task with the help of GUI, the user moves from one layout entity to another.

Appropriateness of the interface can be shown by absolute and relative positions of each layout entities, frequency with which layout entity is used, and the cost of changeover from one layout entity to another.

Cohesion metric for user interface measures the connection among the onscreen contents. Cohesion for user interface becomes high when content presented on the screen is from a single major data object (defined in the analysis model). On the other hand, if content presented on the screen is from different data objects, then cohesion for user interface is low.

In addition to these metrics, the direct measure of user interface interaction focuses on activities like measurement of time required in completing specific activity, time required in recovering from an error condition, counts of specific operation, text density, and text size. Once all these measures are collected, they are organized to form meaningful user interface metrics, which can help in improving the quality of the user interface.

Metrics for Object-oriented Design

In order to develop metrics for object-oriented (OO) design, nine distinct and measurable characteristics of OO design are considered, which are listed below.

- **Complexity:** Determined by assessing how classes are related to each other
- **Coupling:** Defined as the physical connection between OO design elements
- **Sufficiency:** Defined as the degree to which an abstraction possesses the features required of it
- **Cohesion:** Determined by analyzing the degree to which a set of properties that the class possesses is part of the problem domain or design domain
- **Primitiveness:** Indicates the degree to which the operation is atomic
- **Similarity:** Indicates similarity between two or more classes in terms of their structure, function, behavior, or purpose
- **Volatility:** Defined as the probability of occurrence of change in the OO design
- **Size:** Defined with the help of four different views, namely, population, volume, length, and functionality. Population is measured by calculating the total number of OO entities, which can be in the form of classes or operations. Volume measures are collected dynamically at any given point of time. Length is a measure of interconnected designs such as depth of inheritance tree. Functionality indicates the value rendered to the user by the OO application.

Metrics for Coding

Halstead proposed the first analytic laws for computer science by using a set of primitive measures, which can be derived once the design phase is complete and code is generated. These measures are listed below.

n_l = number of distinct operators in a program

n_2 = number of distinct operands in a program

N_1 = total number of operators

N_2= total number of operands.

By using these measures, Halstead developed an expression for overall program length, program volume, program difficulty, development effort, and so on.

Program length (N) can be calculated by using the following equation.

$N = n_1 \log_2 n_l + n_2 \log_2 n_2$.

Program volume (V) can be calculated by using the following equation.

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

Note that program volume depends on the programming language used and represents the volume of information (in bits) required to specify a program. Volume ratio (L) can be calculated by using the following equation.

L = Volume of the most compact form of a program / Volume of the actual program

Where, value of L must be less than 1. Volume ratio can also be calculated by using the following equation.

$$L = (2/n_1)* (n_2/N_2).$$

Program difficulty level (D) and effort (E)can be calculated by using the following equations.

$$D = (n_1/2)*(N_2/n_2).$$

$$E = D * V.$$

Metrics for Software Testing

Majority of the metrics used for testing focus on testing process rather than the technical characteristics of test. Generally, testers use metrics for analysis, design, and coding to guide them in design and execution of test cases. Function point can be effectively used to estimate testing effort. Various characteristics like errors discovered, number of test cases needed, testing effort, and so on can be determined by estimating the number of function points in the current project and comparing them with any previous project.

Metrics used for architectural design can be used to indicate how integration testing can be carried out. In addition, cyclomatic complexity can be used effectively as a metric in the basis-path testing to determine the number of testcases needed.

Halstead measures can be used to derive metrics for testing effort. By using program volume (V) and program level (PL),Halstead effort (e)can be calculated by the following equations.

$$e = V/ PL$$

Where

$$PL = 1/ [(n_1/2) * (N_2/n_2)] \quad \dots (1)$$

For a particular module (z), the percentage of overall testing effort allocated can be calculated by the following equation.

Percentage of testing effort (z) = $e(z)/\sum e(i)$

Where, e(z) is calculated for module z with the help of equation (1). Summation in the denominator is the sum ofHalstead effort (e) in all the modules of the system.

For developing metrics for object-oriented (OO) testing, different types of design metrics that have a direct impacton the testability of object-oriented system are considered. While developing metrics for OO testing, inheritance and encapsulation are also considered. A set of metrics proposed for OO testing is listed below.

- **Lack of cohesion in methods (LCOM):** This indicates the number of states to be tested. LCOM indicatesthe number of methods that access one or more same attributes. The value of LCOM is 0, if no methods access the same attributes. As the value of LCOM increases, more states need to be tested.
- **Percent public and protected (PAP):** This shows the number of class attributes, which are public or protected. Probability of adverse effects among classes increases with increase in value of PAP as public and protected attributes lead to potentially higher coupling.
- **Public access to data members (PAD):** This shows the number of classes that can access attributes of another class. Adverse effects among classes increase as the value of PAD increases.
- **Number of root classes (NOR):** This specifies the number of different class hierarchies, which are described in the design model. Testing effort increases with increase in NOR.
- **Fan-in (FIN):** This indicates multiple inheritances. If value of FIN is greater than 1, it indicates that the class inherits its attributes and operations from many root classes. Note that this situation (where FIN> 1) should be avoided.

Metrics for Software Maintenance

For the maintenance activities, metrics have been designed explicitly. IEEE have proposed Software Maturity Index (SMI), which provides indications relating to the stability of software product. For calculating SMI, following parameters are considered.

- Number of modules in current release (M_T)
- Number of modules that have been changed in the current release (F_e)
- Number of modules that have been added in the current release (F_a)
- Number of modules that have been deleted from the current release (F_d)

Once all the parameters are known, SMI can be calculated by using the following equation.

$$SMI = [M_T – (F_a + F_e + F_d)]/M_T.$$

Note that a product begins to stabilize as SMI reaches 1.0. SMI can also be used as a metric for planning softwaremaintenance activities by developing empirical models in order to know the effort required for maintenance.

Products metrics

Software Measurement: A measurement is a manifestation of the size, quantity, amount or dimension of a particular attributes of a product or process.

It is an authority within software engineering. Software measurement process is defined and governed by ISO Standard.

Need of Software Measurement:

Software is measured to:

1. Create the quality of the current product or process.
2. Anticipate future qualities of the product or process.
3. Enhance the quality of a product or process.
4. Regulate the state of the project in relation to budget and schedule.

Classification of Software Measurement:

There are 2 types of software measurement:

1. **Direct Measurement:**
   In direct measurement the product, process or thing is measured directly using standard scale.
2. **Indirect Measurement:**
   In indirect measurement the quantity or quality to be measured is measured using related parameter i.e. by use of reference.

Metrics:

A metrics is a measurement of the level that any impute belongs to a system product or process. There are 4 functions related to software metrics:

1. Planning
2. Organizing
3. Controlling
4. Improving

Characteristics of software Metrics:

1. **Quantitative:**
   Metrics must possess quantitative nature. It means metrics can be expressed in values.
2. **Understandable:**
   Metric computation should be easily understood, the method of computing metric should be clearly defined.
3. **Applicability:**
   Metrics should be applicable in the initial phases of development of the software.
4. **Repeatable:**
   The metric values should be same when measured repeatedly and consistent in nature.
5. **Economical:**
   Computation of metric should be economical.
6. **Language Independent:**
   Metrics should not depend on any programming language.

Metrics for software quality:

Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.

Software quality metrics can be further divided into three categories −

- Product quality metrics
- In-process quality metrics
- Maintenance quality metrics

Product Quality Metrics

This metrics include the following −

- Mean Time to Failure
- Defect Density
- Customer Problems
- Customer Satisfaction

Mean Time to Failure

It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

Defect Density

It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

Customer Problems

It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.

The problems metric is usually expressed in terms of **Problems per User-Month (PUM)**.

PUM = Total Problems that customers reported (true defect and non-defect oriented problems) for a time period + Total number of license months of the software during the period

Where,

Number of license-month of the software = Number of install license of the software × Number of months in the calculation period

PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year.

Customer Satisfaction

Customer satisfaction is often measured by customer survey data through the five-point scale −

- Very satisfied
- Satisfied
- Neutral
- Dissatisfied
- Very dissatisfied

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example −

- Percent of completely satisfied customers
- Percent of satisfied customers
- Percent of dis-satisfied customers
- Percent of non-satisfied customers

Usually, this percent satisfaction is used.

In-process Quality Metrics

In-process quality metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes −

- Defect density during machine testing
- Defect arrival pattern during machine testing
- Phase-based defect removal pattern
- Defect removal effectiveness

Defect density during machine testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort.

This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect arrival pattern during machine testing

The overall defect density during testing will provide only the summary of the defects. The pattern of defect arrivals gives more information about different quality levels in the field. It includes the following −

- The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.
- The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.
- The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

Phase-based defect removal pattern

This is an extension of the defect density metric during testing. In addition to testing, it tracks the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.

Because a large percentage of programming defects is related to design problems, conducting formal reviews, or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management.

Defect removal effectiveness

It can be defined as follows –

$$DRE = \frac{\text{Defect removed during a development phase}}{\text{Defects latent in the product}} \times 100\%$$

This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called **early defect removal** when used for the front-end and **phase effectiveness** for specific phases. The higher the value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.

Maintenance Quality Metrics

Although much cannot be done to alter the quality of the product during this phase, following are the fixes that can be carried out to eliminate the defects as soon as possible with excellent fix quality.

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix backlog and backlog management index

Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.

Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.

$$BMI = \frac{Number\ of\ problems\ closed\ during\ the\ month}{Number\ of\ problems\ arrived\ during\ the\ month} \times 100\%$$

If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

Fix response time and fix responsiveness

The fix response time metric is usually calculated as the mean time of all problems from open to close. Short fixresponse time leads to customer satisfaction.

The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability tomeet one's commitment to the customer.

Percent delinquent fixes

It is calculated as follows −

$$Percent\ Delinquent\ Fixes = \frac{Number\ of\ fixes\ that\ exceeded\ the\ response\ time\ criteria\ by\ ceverity\ level}{Number\ of\ fixes\ delivered\ in\ a\ specified\ time} \times 100\%$$

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. Formission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.

A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure; the second is a process measure. The difference between the two dates is the latent period of the defective fix.

Usually the longer the latency, the more will be the customers that get affected. If the number of defects is large, then the small value of the percentage metric will show an optimistic picture. The quality goal for the maintenanceprocess, of course, is zero defective fixes without delinquency.