# TEST DESIGN

## TABLE OF CONTENTS

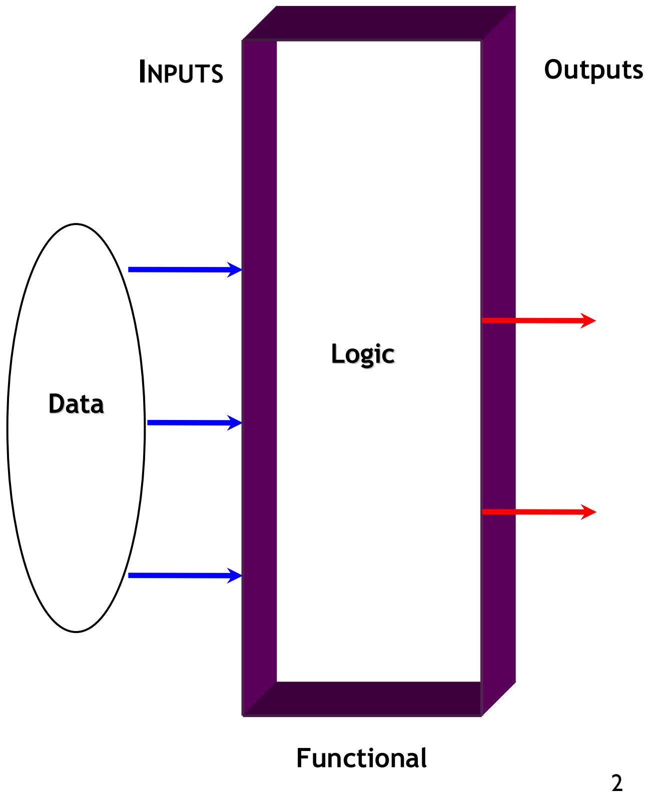
## MODULE – TEST DESIGN (BLACKBOX TECHNIQUES)

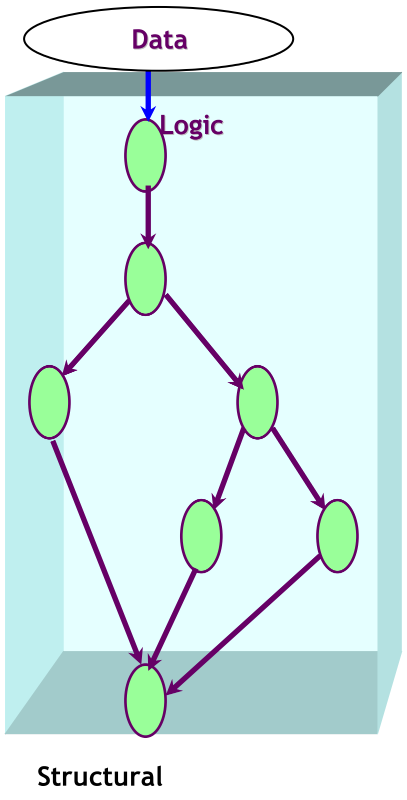
### TERMINOLOGY/DEFINITIONS

| | |
|---|---|
| TS - Test scenario | Test scenario is a thread of operational use that has a path from beginning to end. |
| TC - Test case | Test case is a set of inputs and outputs along with the expected result that satisfy the scenario. |
| Box model | Box model is used to understand the input, output and business logic of the feature |
| Decision table | Decision table is high level black box technique to generate scenario. |
| Exhaustive technique | Exhaustive technique is black box technique which combines all values exhaustively to generate test cases. |
| Test data | Test data is input data which is used for executing test cases. |
| Black-box technique | It is a method of software testing that tests the functionality of an application. Test cases are built around specifications and requirements. These tests can be functional or non-functional. The test designer selects valid and invalid inputs and determines the correct output. |
| Decision tables | It is a table which associates conditions with actions to perform. |
| All-pairs testing | It is a combinatorial software testing method that tests all possible discrete combinations of those parameters. |
| State transition table | It is a table showing what state a finite semi-automaton or finite state machine will move to, based on the current state and other inputs. |
| Equivalence partitioning | It is a software testing technique that divides the input data of a software unit into partitions of data from which test cases can be derived. This technique tries to define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed. |
| Boundary value analysis | Boundary conditions are those situations at the edge of the planned operational limits of the software. Test valid data just inside the boundary, test last possible data and test the invalid data just outside the boundary. |

### BLACK BOX AND WHITE BOX TECHNIQUES

Black box technique is a strategy used for designing the test cases based on the specifications. This does not require any knowledge of the internal logic of the system to develop test cases. Conversely white box technique is a strategy, which uses the internal structure of the system to develop test cases. It requires the knowledge of the structure or internal logic of the system to develop hypothetical test cases. To effectively test systems, we need to use both methods. Fig I shows pictorial representation of these two techniques.





The table below enumerates the advantages and disadvantages of both the methods discussed above:

| **Black Box Technique** | | **White Box Technique** | |
|---|---|---|---|
| Advantages | Disadvantages | Advantages | Disadvantages |
| Simulates actual system usage. | Potential of missing logical errors in the software. | Can test structure of the software | Does not ensure that you have met the user requirements |
| Makes no system structure assumption. | Possibility of redundant testing. | Test the software where you wouldn't think if you performed only functional testing. | This may not mimic real-world situations |

## BLACK BOX TECHNIQUE

It is a technique wherein the tester is unaware of the internal structure of the system, in other words testing is carried to reveal the behavioral details of the software system. In functional testing the test case is designed considering only the input and output as a basis.

Whenever there is an incorrect or missing specification in a software program or system, then that can be detected using black box technique or functional test technique. White box technique fails to uncover these types of defects. Black box technique is all about testing the program or product without knowing the internal structure of it. Or in other words the behavior of the system to particular set of inputs/outputs can be found using this technique.

## WHITE BOX TECHNIQUE

Structural test techniques otherwise known as white box techniques puts forward another test case design strategy where the tester is aware of the internal logic structure or coding of the software system. Testing using white box techniques is done to ensure that the internal components of a program work properly.

It deals with adequacy and coverage of testing in a program. White box or structural test techniques allow for detection of extra functionality implemented that has not been stated in the specification. Black box techniques will fail to detect these "extra features" as these have not been specified.

### TEST SCENARIO

Sets of test cases that ensure that the business process flows are tested from end to end. They may be independent tests or a series of tests that follow each other, each dependent on the output of the previous one.

Scenarios are long at system test level while they are short at the unit test level.
A test scenario can be: (a) Positive or negative (b) Valid or invalid

### TEST CASE

Test case is a set of input(s)-output(s) along with the expected result that satisfy a scenario. A test case in a practical sense is a test-related item, which contains the following information:
A set of test inputs – These are data items received from an external source by the code under test.
Execution condition – These are conditions required for running the test.
Expected outputs – These are specified results to be produced by the code under test.
The above description specifies minimum information that should be found in a test case. An organization may decide that additional information should be included in a test case to increase its value as a reusable object, or to provide more detailed information to testers and developers. As an example, a test objective component could be included to express test goals such as to execute a particular group of code statements or check that a given requirement has been satisfied. Developers, testes, and/or software quality assurance staff should be involved in designing a test case specification that precisely describes the contents of each test case. The contents and its format should appear in test documentation standards for the organization.

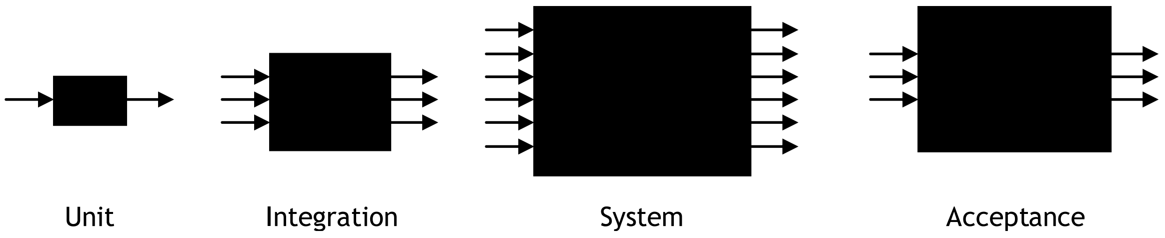
**Example:**
**Login screen: Unit level**
TS1: Ensure that an invalid user is not allowed to log in <Negative scenario>
TC1: Invalid name – Name is NULL
TC2: Invalid name – Name is too long
TC3: Invalid name – Name contains illegal characters
TC4: Invalid name – Name is not a registered user name

## BLACK BOX TECHNIQUES

Black box technique is a technique of designing test cases that focuses on the functional requirements of the software system. That is, it enables the software engineer to derive sets of input conditions that will fully exercise all the functional requirements of the system. Black box technique is not an alternative to white box testing; rather it is a complementary approach that is likely to uncover a different class of errors than those found by white box methods. Because black box technique purposely disregards control structure, attention is focused on the information domain.

In this, testing is based on the functional system requirements that include both data and business logic. Here we give a set of input data to the system in the form of test cases and check whether the expected results match with the actual outcome of the system.

Black box techniques can be applied at any levels of system development – unit, integration, system and acceptance.



As we move up in size from module to subsystem to system the box gets larger with more complex inputs and more complex outputs, but the approach remains the same. Also as we move up we are forced to use the black box approach, as there are more complex paths through the software under test to perform white box testing.

**It attempts to find errors in the following categories-**
- Incorrect or missing functions and specifications
- Interface errors
- Errors in data structures or external database access
- Performance errors and
- Initialization and terminations

**Tests are designed to answer the following question:**
- How is the functional validity tested?
- What classes of inputs will make a good test case?
- Is the system particularly sensitive to certain types of input values?
- How are the boundaries of the data class isolated?
- What data rate and data volume can the system tolerate?
- What effects will the specific combinations of data have on system operations?

**By applying black box techniques we derive a set of test cases that satisfy the following criteria:**
- Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing.
- Test cases that tell us something about the presence or absence of class of errors, rather than errors associated only with the specific test at hand.

### KNOWLEDGE SOURCES

Using the black box approach the tester considers the software under test as an opaque object. Here the tester has the knowledge of what the system does and doesn't have anything to do with the internal structure of the system. The tester provides the specified inputs to the system under test, runs the tests and then determines whether the output produced is equivalent to those in the specification. Since the black box considers only the software behavior and functionality, it is often called functional or specification based testing.

In order to carry out Black box technique the knowledge sources required are as follows:
- Customer Requirements document at the Acceptance test level
- Software Requirements document at the System test level
- High level design document at the Integration Test level
- Low level design document at the Unit Test level
- Specifications
- Domain knowledge
- Defect analysis data

## ISSUES IN BLACK BOX TECHNIQUE

When using Black box technique the tester can never be sure of how much of the software under test has been tested. Some execution paths may never be executed. To find every defect using black box technique, the tester may have to generate all possible combinations of input data, both valid and invalid, which is almost impossible.

## METHODS IN BLACK BOX TECHNIQUE

The Black box technique is based on the data structure i.e., data and the business logic of the application under test. The important types are as discussed below:

1. Equivalence class partitioning
2. Boundary value analysis
3. Special value/ Error guessing
4. Error based
5. Input/Output domain
6. Decision table technique
7. State diagram technique
8. Flowchart technique

## EQUIVALENCE CLASS PARTITIONING

Equivalence class partitioning is a technique used to reduce the number of test cases to manageable level while still maintaining reasonable test coverage. This simple technique is used intuitively by almost all testers, even though they may not be aware of it as a formal test design method.

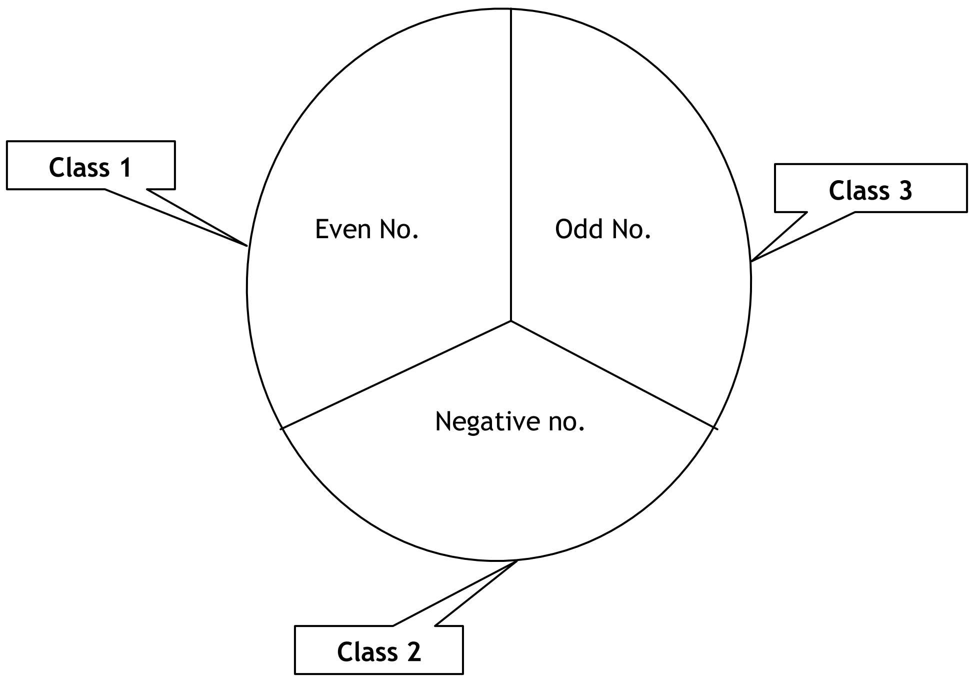
An equivalence class or equivalence partition is a set of test cases that test the same thing or reveals the same bug. Here we partition the data under test into equivalence classes whose intention is to get completeness but at the same time avoid redundancy.
Traditional view is to partition the data into two classes namely valid and invalid. When looking for EC, normally we think about ways to group similar input(s), similar output(s) and similar operation of software.

**How to apply the technique of EC:**
The EC consists of a set of data that is treated the same or that should produce the same result. Any data value within a class is equivalent to any other value in terms of testing.

The steps for using EC testing are simple:
Identify the equivalence classes.
Create test cases for each equivalence class.

**Example:**
**A set of numbers –50 to 50 is given. How many test cases can be generated using EC if we had to divide them into even, odd and negative numbers?**

In order to write the test cases for a set of numbers that are even, odd and negative we first need to identify the equivalence class. Refer to below figure:



Once the class is identified, the test case design based on each equivalence class will now be easier. So based on the classes formed using EC, there will be 3 test cases from each class.

**Advantages of equivalence classes:**
It eliminates the need for exhaustive testing which is not otherwise feasible.
It guides a tester in selecting a subset of inputs with a high probability of detecting a defect.
It allows a tester to cover a large domain of inputs/outputs with a smaller subset selected from an equivalence class.

## BOUNDARY VALUE ANALYSIS

Boundary conditions are those situations at the edge of the planned operational limits of the software. Test valid data just inside the boundary, test the last possible data and test the invalid data just outside the boundary. Boundary types can be Numeric, Character, Position, Quantity, Speed, Location, Size, etc.

With experience, testers soon realize that many defects occur directly on, and above and below the edges of the equivalence classes. Test cases that consider these boundaries on both the input and output spaces are very often valuable in revealing defects.

The thumb-rules described below are useful for getting started with boundary value analysis:
If an input condition for the software-under-test is specified as a range of values, develop valid test cases for the ends of the range, below the ends of the range and invalid test cases for the possibilities just above the range.
If an input condition is specified as a number of values, develop test cases for the minimum and maximum numbers as well as invalid test cases that include one lesser and one greater than the maximum and minimum.
If the input or output of the software under test is an ordered set, such as a table or a linear list, then develop test cases that focus on first and last elements of the set.

**Example:**
Given a login screen that has the following character length limit for the username and password. No special characters or numerals are allowed. Applying BVA, how many test cases can be generated?

| | | |
|---|---|---|
| Username: | Min char = 4 | Max char = 15 |
| Password: | Min char = 3 | Max char = 12 |

Username: ____________________

Min = 4 Max = 15

Password:

____________________

**Hint:**
Applying BVA to check username:
Check for username length 4, 15, 5 and 14 for positive test cases. Check for username length 3 and 16 for negative test cases. Total number of values obtained by BVA is 6.

## SPECIAL VALUE OR ERROR GUESSING

Selection of test data is on the basis of features of a function and tester uses domain knowledge to come up with different test cases. Designing test cases using this approach is based on tester's or developers past experience with the software-under-test, and their intuition as to where defects may lurk in the code. Error guessing is an ad hoc approach to test design in most cases.

## ERROR BASED

In this method test cases are generated based on:
- History of the programmers
- Complexity of the program
- Knowledge of error-prone syntactic constructs
- Guessing errors based on data type is also possible in this method

## CAUSE-EFFECT GRAPHING OR DECISION TABLE

This technique can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification. The specification must be transformed into a graph that resembles a digital logic circuit for which knowledge of Boolean logic is necessary.

| | | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Causes | C1 | F | | | | | |
| | C2 | T | | | | | |
| | C3 | T | | | | | |
| | C4 | T | | | | | |
| Effects | E1 | | | | | | |
| | E2 | ● | | | | | |

A major weakness with the equivalence class partitioning is that it does not allow testers to combine conditions. Combinations can be converted in some cases by test cases generated from the classes. A decision table helps to overcome these problems effectively by allowing cause-effect combinations to develop test cases. Fig 3 shows a sample decision table.

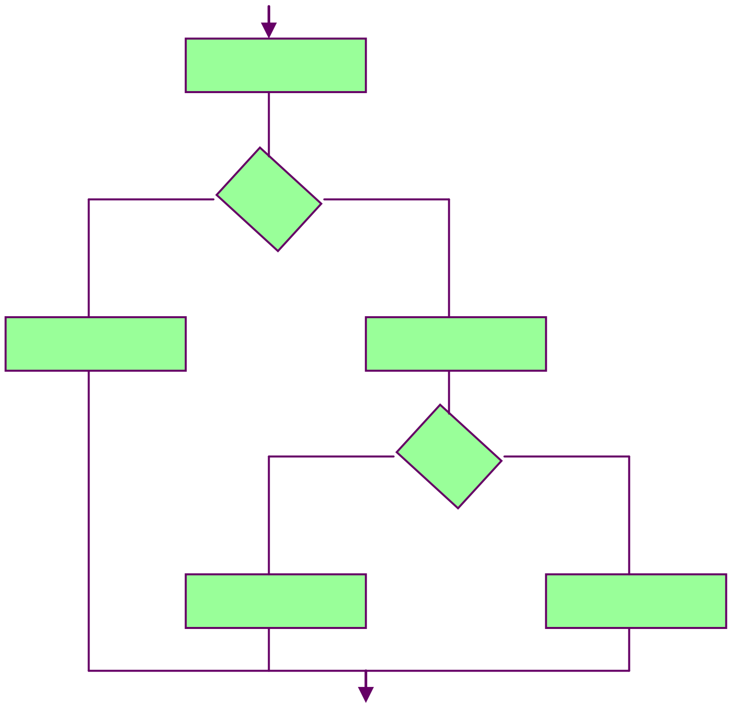
The steps involved in developing test cases are as listed below:

- Decompose the specification of the complex software component into lower level units.
- For each specification unit the tester needs to identify causes and their effects. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation. The logical relationship between the causes and their corresponding effects is to be determined.
- From the above information, a Boolean cause-and-effect graph is created. C1, C2, C3, etc represents the causes and E1, E2, E3, etc represents causes and their effects respectively.
- The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
- The graph is then converted into decision table.
- The columns in the decision table are then transformed into test cases.

**Reducing the decision table:**
One goal of testing is to reduce number of test cases yet still provide the opportunity to find problems.
It is possible to merge rules whose resulting actions are identical and thus simplify the logical expression. The theory behind this comes from hardware logic design, which employs Boolean algebra to manipulate and evaluate logical expression. If the logical expression can be written in an equivalent form using fewer variables, then the corresponding hardware circuit will be simpler.

## FLOW CHART

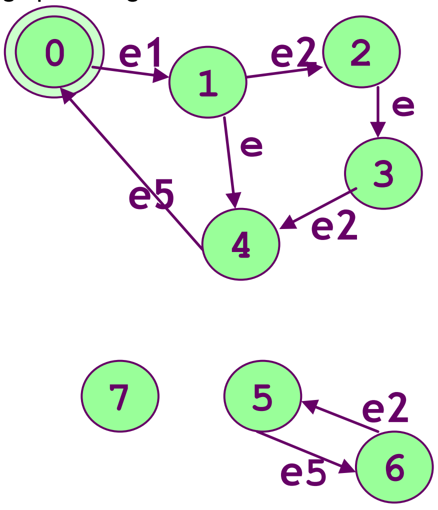
It is a simple notation for the representation of logical control flow in a program or program control structure. Any procedural design representation can be translated into a flow chart. Based on the flow chart, test cases are generated using boundary value analysis, equivalence partitioning, error guessing, etc. A simple flow chart is shown in the figure below:



Here, a box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control. The sequence is represented as two processing boxes connected by a line of control.

## STATE TRANSITION TESTING

It is based on the concepts of the state and finite state machines, and allows the tester to view the developing software in terms of its states, transition between states and the inputs, and events that trigger state changes where state is an internal configuration of the system or a component. A finite state machine is an abstract machine that can be represented by a state graph having finite number of states and finite number of transitions between states.



The above figure shows a sample state machine diagram. The arrows represent events and round shapes denote the state. Note that transition from one state to another takes place only if the event takes place.

The main features of state transition testing are:
- This is event based business logic.
- It covers all paths generated in test cases.
- For every event we can generate test cases using boundary value analysis, equivalence classes, etc.
- It helps in discovering the incomplete specification.

## ORTHOGONAL ARRAYS

An orthogonal array is a two dimensional array of numbers that has this interesting property - choose any two columns in the array. All the pairwise combinations of its values will occur in every column pair.

The origin of orthogonal arrays can be tracked back to Euler, the great mathematician, in the guise of Latin Squares. Genichi Taguchi has popularized their use in hardware testing.

### Using orthogonal arrays

The process of using orthogonal arrays to select pairwise subsets for testing is:
- Identify the variables
- Determine the number of choices for each variable
- Locate an orthogonal array, which has a column for each variable and values within the columns that correspond to the choices for each variable.
- Map the test problem onto the orthogonal array
- Construct the test cases

Consider the numbers 1 and 2. How many pair combinations (combinations taken two at a time) of '1' and '2' exist? {1,1}, {1,2}, {2,1} and {2,2}.

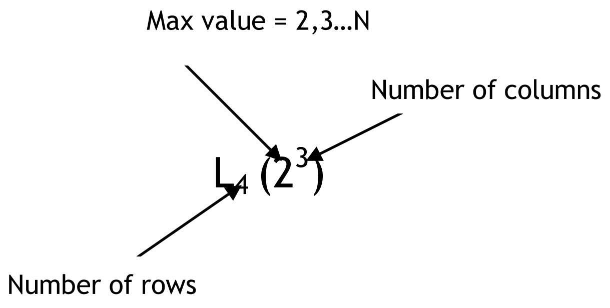
Let's examine an $L_4$ ($2^3$) array

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | **1** | **1** | **1** |
| 2 | **1** | **2** | **2** |
| 3 | **2** | **1** | **2** |
| 4 | **2** | **2** | **1** |

$L_4$ ($2^3$) orthogonal array

The gray column headings and row numbers are not part of the orthogonal array but are included for convenience in referencing the cells. Examine columns 1 and 2 – do the four combinations of 1 and 2 all appear in that column pair? Yes, and in the order listed earlier. Now examine columns 1 and 3 – do the four combinations of 1 and 2 appear in that column pair? Yes, although in a different order. Finally, examine columns 2 and 3 – do the four combinations appear in that column pair also? Yes they do. The $L_4$ ($2^3$) is orthogonal; that is, choose any two columns, all the pairwise combinations will occur in all the column pairs.

A note about the curious (but standard) notation: $L_4$ means an orthogonal array with four rows, ($2^3$) is not an exponent. It means that the array has three columns, each with either 1 or a 2.



## USE CASE TESTING

Use case approach is a test case design technique that is applicable only at system level. The use cases exercises a system's functionalities from start to finish by testing each of its individual transactions. Defining the transactions that a system processes is a vital part of the requirements definition process. Like other test case design techniques the use cases hold a wealth of information useful to testers.

Use case can be defined as a scenario that describes the use of a system by an actor to accomplish a specific goal. By actor we mean a user playing a role with respect to the system seeking to use the system to accomplish something worthwhile within a particular context. Actors are generally people although other systems may also be actors. A scenario is a sequence of steps that describe the interactions between the actor and the system. Note that the use case is defined from the perspective of a user, not the system. Also note that the internal working of the system, while vital, is not part of the use case definition. The set of use cases makes up the functional part of a system.

The value of the use cases is that they provide the basis for identifying a system's key internal components, structures, databases and relationships. They also serve as the foundation for developing test cases at the system as well as acceptance level.

It is always advisable to subject each use case through an inspection process before it is implemented. To test the implementation, the basic rule is to create at least one test case for the main test case scenario and at least one test case for each extension.

Since use cases do not specify input data, the tester must select it. It is important to consider the risk of the transaction and its variants under test. To create test cases, start with normal data for the most often used transactions. Then move to boundary values and invalid data.

# TEST DESIGN

The template for use case:

| Use case component | Description | |
|---|---|---|
| Use case no. Or identifier | A unique identifier for this use case | |
| Use case name | The name should be the goal stated as a short active verb phrase | |
| Goal in context | A more detailed statement of the goal(if necessary) | |
| Scope | Corporate \| System \| Subsystem | |
| Level | Summary \| Primary Task \| Sub-function | |
| Primary actor | Role name or the description of the primary actor | |
| Preconditions | The required state of the system before the use case is triggered | |
| Success end conditions | The state of the system upon successful completion of this issue | |
| Failed end conditions | The state of the system if the use case cannot execute to completion | |
| Trigger | The action that initiates the execution of the use case | |
| Main success scenario | Step | Action |
| | 1 | |
| | 2 | |
| Extensions | Conditions under which the main success scenario will vary and a description of those variations | |
| Sub-variations | Variations that do not affect the main flow but that must be considered | |
| Priority | Criticality | |
| Response time | Time available to execute this use case | |
| Frequency | How often this use case is executed | |
| Channels to primary actor | Interactive \| File \| Database \|… | |
| Secondary actors | Other actors needed to accomplish this use case | |
| Channels to secondary actors | Interactive \| File \| Database \|… | |
| Date due | Schedule information | |
| Completeness level | Use case identified (0.1) \| Main scenario defined (0.5) \| All extensions defined (0.8) \| All fields complete (1.0) | |
| Open issues | Unresolved issues awaiting decisions | |

**Sample Use Case:** Handle a claim (Business)
**Use Case Identifier:** HAC1100
**Use Case Name:** Handle a Claim
**Scope**: Insurance company operations
**Level**: Summary
**Primary Actor:** Insurance claimer
**Preconditions:** A loss has occurred
**Release**: 1st
**Status**: Ready for review
**Revision**: Current
**Context of use**: Customer wants to get paid for an incident
**Trigger**: A claim is reported to insurance company

**Main success scenario:**

1. Customer reports a claim (phone, fax, paper or email) to clerk
2. Clerk finds the policy, registers loss in system, and assigns an Adjuster
3. Adjuster investigates the claim and updates the claim with additional information
4. Adjuster enters progress notes over time
5. Adjuster corrects entries and sets monies aside over time
6. Adjuster receives documentation including bills throughout the life of the claim and enters bills
7. Adjuster evaluates damages for claim and documents the negotiation process in system
8. Adjuster settles and closes claim in system
9. System purges claim 6 months after close
10. System archives claim after time period

**Extensions:**

1a. Submitted data is incomplete

1a1. Insurance Company requests for missing information

1a2. Claimants supplies missing information

1a2a. Claimants does not supply information within time period

1a2a1. Adjuster closes claim in system

Frequency of occurrence: Not mentioned
Success guarantee: claim is closed, settled and archived
Minimal guarantee: Claimed closed but may be reopened later
Stakeholders and Interests:
The company – makes smallest accurate settlement
Customer – get largest settlement
Department of Insurance – ensure correct procedure
Business rules: Not mentioned
Data Descriptions: None
UI links: None
Open Issues: None

**References:**

KIT, EDWARD, Software Testing In the Real World, Pearson Education, 2003.
PRESSMAN, ROGER S., Software Engineering A Practitioner's Approach, McGraw-Hill, 2001.
Burnstein, ILENE. , Practical Software Testing, Springer, 2002.
Tamres, Louis. , Software Testing, Pearson Education Limited, 2002.

## MODULE EXERCISES

1. What are the various test design techniques available from Black box perspective?
2. Design test cases for quality levels 1, 2, 3, 4, 5, 6, 7, 8, 9 for features of SugarCRM application.

## POINTS TO PONDER

1. Can black box techniques be used to design test cases for any kind of application in any domain? State reasons why.

2. "It is impossible to prove that testing is adequate if only black box techniques have been applied to design test cases" - Why?

3. Is it possible that the test cases generated using different techniques like flowchart or a decision table will be different? State reason in either case.

4. Is one technique (Flowchart or Decision table) superior to the other? Discuss.

5. Given an input that is a list box for color where the drop-down choices are {Red, Blue, Yellow}, what are the equivalence classes and the corresponding test values?

6. Can the black box techniques be applied at any level of testing? State reasons.

7. What should be the ratio of +ve and –ve test cases to ensure effective testing?

## MODULE – TEST DESIGN (WHITEBOX TECHNIQUES)

## TERMINOLOGY/DEFINITIONS

| | |
|---|---|
| Code coverage | It describes the degree to which the source code of a program has been tested. Creating tests to cause all statements, branches and paths in the program to be executed at least once. |
| Control flow | The order in which the individual statements, instructions, or function calls of a program are executed or evaluated. |
| Data flow | The path of defining the data, the course it takes to go through various modifications and usage in different instances. |
| Cyclomatic complexity | The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. |
| Statement coverage | Execute each node or statement in the program. |
| Decision coverage | Execute every edge in the program. |
| Multiple condition coverage | This criteria requires that all combinations of conditions inside each decision are tested. |
| Path coverage | Execution of every possible route through a given part of the code. |
| Loop coverage | The number of times the loop in a program is executed. |

### WHITEBOX TECHNIQUES

"Bugs lurk in corner and congregate in the boundaries." White box technique is far more likely to uncover them.

White box technique is a testing technique where knowledge of the internal structure and logic of the system is necessary to develop hypothetical test cases. It's sometimes called structural testing or glass box testing. It's also a test case design method that uses the control structure of the procedural design to derive test cases.

If a software development team creates a block of code that allows the system to process information in a certain way, a test team would verify this structurally by reading the code and given the systems' structure, see if the code could work reasonably. If they feel it could, they would plug the code into the system and run an application to structurally validate the system.

White box test designs are derived from the internal design specification or the code. The knowledge needed for white box test design approach often becomes available during the detailed design of development cycle.

#### OBJECTIVE OF WHITE BOX TECHNIQUES

Using white box techniques, software engineer can improve the test cases that will

- Guarantee that all independent paths within a module have been executed at least once
- Exercise all logical decisions on their true and false sides
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to assure their validity

#### WHY USE WHITE BOX TECHNIQUE?

A reasonable question that arises in our mind is why should we carry out white box technique on a software product?
Answer lies in the following points:
Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed - errors tend to creep into our work when we design and implement functions, conditions and control that are out of the mainstream. Everyday processing tends to be well understood while special case processing tends to fall in cracks.

We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis – our assumptions about the logical flow of a program and data may lead us to make design errors that are uncovered only once path testing commences.

Typographical errors are random – when a program is translated into its programming language source code it is likely that some typing errors will occur. It is likely that a typo will exist on obscure logical paths on a mainstream path.

#### ASPECTS OF CODE TO CONSIDER

White box technique of software is predicated on a close examination of procedural detail. White box test cases exercise specific sets of conditions and / or loops tests logical paths through the software. The "Status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

## TEST DESIGN

At first glance it would seem that very thorough white box technique would lead to 100% correct program. All we need to do is to define all logical paths. Develop test cases to exercise them and evaluate their results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. For example, the procedural design that might correspond to a 100-line Pascal program with a single loop that may be executed no more than 20 times. There are approximately 10*14 possible paths that may be executed!

To put this number in perspective, we assume that a magic test processor (”magic”, because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it and evaluate the result in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program represented in figure 2. This would, undeniably cause havoc in most development schedules. Exhaustive testing is impossible for large software systems. White box technique should not, however be dismissed as impractical. Limited number of important logical paths can be selected and exercised. Important data structure can be probed for validity.

### TEST ADEQUECY CRITERIA

White box technique is very useful in achieving the test adequacy rather than designing test cases.
The test adequacy criteria are based on the “code coverage analysis” which includes coverage of:

- Statements in software program
- Branches or multiple condition branches those are present in a code
- Exercising program paths from entry to exit
- Execution of specific path segments derived from data flow combinations definitions and uses of variables

The code coverage analysis methods mentioned above are discussed in the section below.

The goal for white box technique is to ensure that internal components of a program are working properly. A common focus is on structural elements such as statements and braches. The tester develops test cases that exercise these structural elements to determine if defects exist in the program structure. By exercising all of these structural elements, the tester hopes to improve the chances of detecting defects.

The testers need a framework for deciding which structural elements to select as a focus of testing, for choosing the appropriate test data and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly. The criteria can be viewed as representing minimal standards for testing a program.

The application scope of adequacy criteria also includes:

- Helping testers to select properties of a program to focus on during tests.
- Helping them to select a test data set for the program based on the selected properties.
- Supporting testers with the development of quantitative objectives for testing.
- Indicating to testers whether or not testing can be stopped for that program.

A program is adequately tested to a given criterion if all the target structural elements have been exercised according to the selected criterion. Using the selected adequacy criterion a tester can terminate testing, when the target structures has been exercised. Adequacy criteria are usually expressed as statements that depict the property or feature of interest, and the conditions under which testing can be stopped.

In addition to statement and branch adequacy criteria, other types of program-based test adequacy criteria in use are the path testing and loop testing wherein, the total paths and all the loops are executed.

### CODE COVERAGE METHODS

Logic based white box test design and use of test data adequacy/coverage concepts provide quantitative coverage goals to the tester. As mentioned in the previous section 1.4 test adequacy criteria and coverage analysis play an important role in testing. The major techniques involved in code coverage analysis are:

**Statement testing:**
In this method every source language statement in the program is executed at least once so that no statements are missed out. Here we need to be concerned about the statements controlled by decisions. The statement testing is usually regarded as the minimum level of coverage in the hierarchy of code coverage and is therefore a weak technique. It is based on - feeling absurd to release a piece of software without having executed every statement.

In statement testing, 100% coverage may be difficult if

- There exists unreachable code
- The exception handling in the code is improper
- The missing statements also create issues

Statement coverage can be calculated as:
Statement coverage = #Statements Executed / #Total Statements

**Branch testing:**
Branch testing is used to exercise the true and false outcomes of every decision in a software program or a code. It is a superset of statement testing where the chances of achieving code coverage is more. The disadvantage of branch testing is that it is weak in handling multiple conditions.

Branch coverage can be found out as:
Branch coverage = #Branches Executed / #Total Branches

**Multiple conditions coverage:**
Multiple conditions coverage is a testing metrics that exercise on all possible combinations of true and false outcomes of conditions within a decision in the control structure of a program. This is a superset of branch testing.

The primary advantage of multiple condition testing is that it tests all feasible combinations of outcomes for each condition with in program.

The drawback of this metrics is that it provides no assistance for choosing data. Another expensive drawback of this coverage is that the decision involving n Boolean conditions has 2n combinations => 2n test cases.

Multiple condition coverage can be calculated as:
Multiple condition coverage = #Conditions Executed / #Total Multiple Conditions

**Path testing:**
Here we can design a set of test cases by which every path in the program (in control flow graph) is executed at least once.

The figure 2 in section 2.1 shows the graphical representation of flow of control in a program. The number of paths in this representation can be found out and we can clearly say that all paths are distinct. Note that the path shown by 1-2-4-6-7-8 is different from that depicted in 1-3-4-5-7-8.
The major disadvantage of path testing is that we cannot achieve 100% path testing, as there are potentially infinite paths in a program, especially if the code is large and complex then path testing is not feasible.

**Modified path testing:**
Modified path testing as the name suggests is a modification of existing path testing and is based on number of distinct paths to be executed, which is in turn based on McCabe complexity number. The McCabe number is given as $V = E – N + 2$, where E is the number of edges in the flow graph and N is the number of nodes.

Testing criterion for modified path testing is as given below:
Every branch is executed at least once
At least V distinct paths must be executed
Choosing the distinct path is not easy and different people may choose different distinct paths

Path coverage is assessed by:
Path coverage = #Paths Executed / #Total Paths

**Loop testing:**
Loop testing strategies focus on detecting common defects associated with loop structures like simple loops, concatenated loops, nested loops, etc. The loop testing makes sure that all the loops in the program have been traversed at least once during testing. Defects in these areas are normally due to poor programming practices or inadequate reviewing.

## WHY USE CODE COVERAGE ANALYSIS?

Code coverage analysis is necessary to satisfy test adequacy criteria and in practice is also used to set testing goals and, to develop and evaluate test data. In the context of coverage analysis, testers often refer to test adequacy criteria as "coverage criteria."
When coverage related testing goal is expressed as a percentage, it is often called "degree of coverage." The planned degree of coverage is usually specified in the test plan and then measured when the tests are actually executed by a coverage tool. The planned degree of coverage is usually specified as 100% if the tester wants to completely satisfy the commonly applied test adequacy or coverage criteria.

Under some circumstances, the planned degree of coverage may be less than 100% possibly due to the following reasons:
1. The nature of the unit.
    - Some statements or branches may not be reachable.
    - The unit may be simple.
2. The lack of resources
    - The time set aside for testing is not adequate to achieve 100% coverage.
    - There are not enough trained testers to complete coverage for all units.
    - There might be lack of tools to support complete coverage.
3. Other project related issues such as timing, scheduling, etc.

The concept of coverage is not only associated with white box technique. Testers also use coverage concepts to support black box technique where a testing goal might be to exercise or cover all functional requirements, all equivalence classes or all system features. In contrast to black box approaches, white box based coverage goals have stronger theoretical and practical support.

The application of coverage analysis is associated with the use of control and data models to represent structural elements and data. The logic elements are based on the flow of control in a code. They are:
- Program statements
- Decisions or branches (these influence the program flow of control)
- Conditions (expression that evaluate true/false and do not contain any other true/false valued expression)
- Combinations of decisions and conditions.
- Paths (node sequence in flow graphs)

All structured programs can be built from three basic primes – sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops).
Primes are nothing but the representation of flow of control in a software program which can take any one form. The figure 1 shows the graphical representation of primes mentioned above.

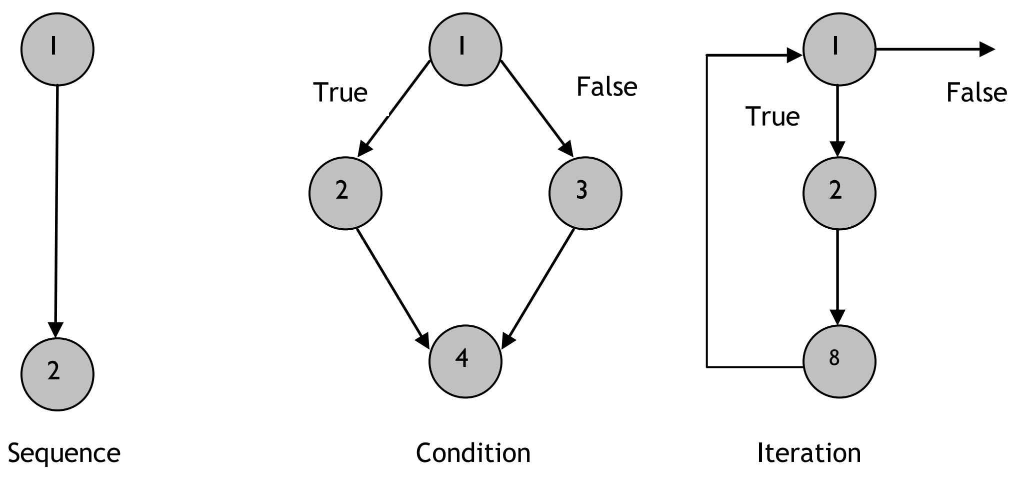


Figure 1

Using the concept of a prime and the ability to use the combinations of primes to develop structured code, a control flow diagram for the software under test can be developed. The tester to evaluate the code with respect to its testability as well as to develop white box test cases can use the flow graph. The direction of the transfer depends upon the outcome of the condition.

## DISADVANTAGES OF WHITE BOX TECHNIQUES

If the numbers of execution paths are very large, then they cannot all be tested. Attempting to test all execution paths is generally as infeasible as testing all input data combinations through black box testing.

The white box technique assumes the control flow is correct or very close to correct. Since the tests are based on the existing paths, nonexistent paths cannot be discovered through white box technique.
The tester must have programming skills to understand and evaluate the software under test. Most of the software testers today don't have this background.

## CODE COMPLEXITY

### CYCLOMATIC COMPLEXITY

Thomas McCabe coined the term cyclomatic complexity in. The cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basic set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

The concept of cyclomatic complexity can be well explained using a flow graph representation of a program as shown below in figure 2.

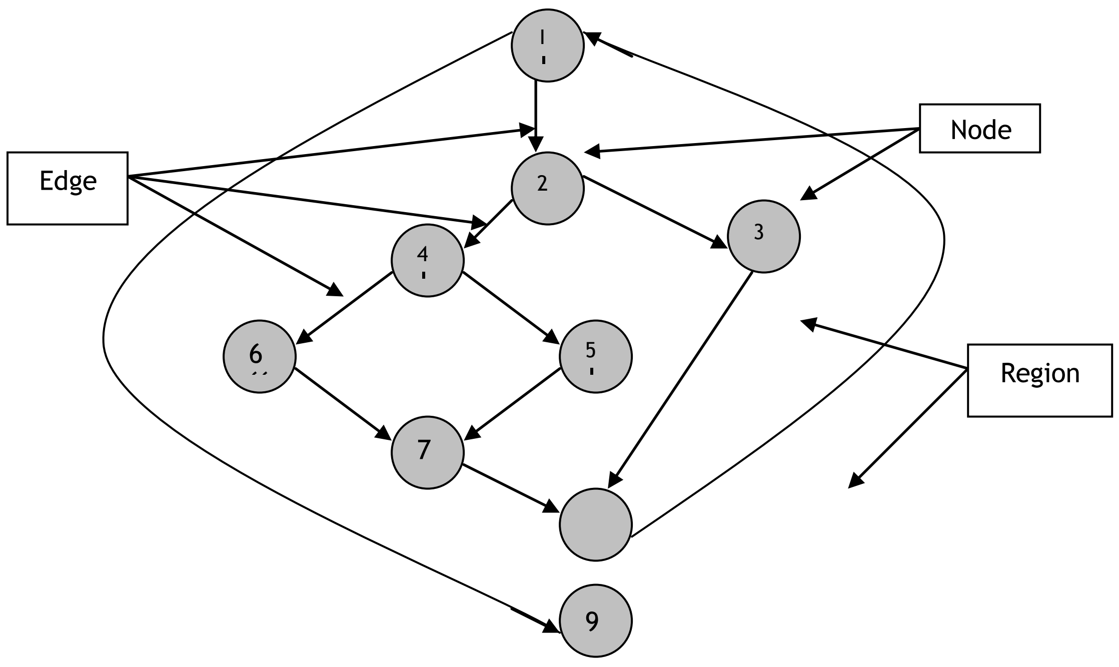


Figure 2

In the diagram above circles denote nodes, and arrows denote edges. Each circle represents a processing task (one or more source code statements) and arrows represent flow of control.

### CODE COMPLEXITY VALUE

McCabe defines a software complexity measure that is based on the cyclomatic complexity of a program graph for a module. Otherwise known as code complexity number, the cyclomatic complexity is a very useful attribute to the tester. The formula proposed by McCabe can be applied to flow graphs where there are no disconnected components. Cyclomatic Complexity is computed in one of the following ways:

1. The number of regions of the flow graph edges corresponds to the cyclomatic complexity.

2. Cyclomatic complexity, V, for a flow graph is defined as
$V = E – N + 2$, where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity, V is also defined as
$V = P + 1$, where P is the number of predicate nodes contained in the flow graph. A predicate node is the one having a condition in it. In the figure 2, example for a predicate node is the circle 1.

For Figure 2 the cyclomatic complexity V = 11 edges – 9 nodes + 2 = 4, using the formula $V = E – N + 2$.
Also V = 3 predicate nodes + 1 = 4, using $V = P + 1$.

### USE OF CYCLOMATIC COMPLEXITY

The code complexity value is useful to programmers and testers in a number of ways, which are enlisted below:
- It provides an approximation on number of test cases that must be developed for branch coverage in a module of structured code.
- It provides an approximation of the testability of module.
- The tester can use the value of V along with past project data to approximate the testing time and resources required to test a software module.
- The complexity value V along with control flow graph give the tester another tool for developing white box test cases using the concept of path.
- It is also a measure of number of so called independent or distinct paths in the graph.

**References:**

PERRY, WILLIAM E., Effective Methods for Software Testing, Wiley 2000.
PRESSMAN, ROGER S., Software Engineering A Practitioner's Approach, McGraw-Hill, 2001.
Burnstein, ILENE. Practical Software Testing, Springer, 2002.
RADICE, RONALD A., Software Inspections, Tata McGraw-Hill, 2003.
C. R. PANDIAN. , Software Metrics, Arunodaya Printers, 2003.

## MODULE EXERCISES

1. What are the various test design techniques available from White box perspective?

## POINTS TO PONDER

1. How many distinct paths does a program with Cyclomatic complexity of 5 have?

2. How can cyclomatic complexity be used to focus on which units to focus on?

3. Given a program as below, which coverage would you prefer?
   Here is a piece of code to calculate price of railway tickets, based on the age of the person traveling.

```
If (5<age<=10) price= 0.5*X
Else
     if (10<age<=18) price=0.75*X
Else
     if (18<age<=50) price=X
Else
     if (50<age<=100) price= 0.5*X
Else
     printf ("Enter correct age")
```

4. Given a program as below, which coverage would you prefer?
   Given two numbers a and b, the code below calculates the difference.

```
If (a > b)
C = a - b
Else c = b – a
```

5. The statement coverage metric for a unit is found to be 100%. Can we be confident that the code has been well tested? If NOT state why?

6. Give examples in which white box technique might give an impression that everything is ok, but black box technique reveals that there are bugs.

7. Is it necessary to have programming skills to apply white box techniques successfully? Discuss.

8. "The actual code is used to design test cases in the white box technique" – Is this statement correct? State reasons either way.

9. One of the measures for test adequacy criteria is code coverage information. Discuss this is do.

10. What do you understand when it is said that the cyclomatic complexity value is 15?

11. If the code coverage is 100%, does that mean testing is complete?

12. If the path coverage for a program that uses dynamic resources is 100%, is it possible to state that all possible defects have been uncovered? Explain in detail.

# MODULE – TEST DESIGN (LSPS TESTING)

## TERMINOLOGY/DEFINITIONS

| | |
|---|---|
| Load test | Check that system is able to handle real-life operations with the stated resources. |
| Stress test | Check that worst load it can handle is well above real-life extreme load. |
| Performance test | Check that key system operations perform within the stated time. |
| Scalability testing | Check that the system is able to handle more load by adding more hardware resources. |
| Reliability test | Check that the system when used in an extended manner is free from failures. |
| Volume test | Check that the system can handle large amounts of data. |
| Configuration test | Check that the system can execute on different HW and SW configurations. |
| Compatibility test | Check that the current system is backward-compatible to its prior versions. |
| Installation test | Check that system can be installed correctly following the installation instructions. |
| Documentation test | Check that the user documentation/on-line help is in line with the software functionality. |
| Compliance test | Check that the software has implemented the applicable standard correctly. |

### LOAD TESTING

A load is a series of inputs that simulates a group of transaction. A transaction is a unit of work as seen from a system user's point of view. Transaction consists of a set of operations that may be performed by a person, software system or a device that is outside the system. For system testing, this transaction we would need a load that would simulate a series of transactions of particular types and lengths arriving simultaneously.
For example, if you were system testing a telecommunication system you would need a load that simulated a series of phone calls (transactions) of particular types and lengths arriving from different locations. A load can be a real load, that is, you could put the system under test to real usage by having actual telephone users connected to it.
Loads generators can also be used which are basically automated test tools for load testing. They will generate test input data for system test. Load generators can be simple tools that output a fixed set of predetermined transactions. They can be complex tools that use statistical patterns to generate input data or simulate complex environments.
Operational profiles are of great help in performing load tests.

### DEVELOPING OPERATIONAL PROFILES

A software-based product's reliability depends on just how a customer will use it. Making a good reliability estimate depends on testing the product as if it were in the field. The operational profile, which is a quantitative characterization of how system will be used, is thus essential in software reliability engineering (SRE). Operational profile theory can be applied to hardware, software and even to the human components. Thus it is applicable to complete systems. The operational profile shows you how to increase productivity and reliability and speed development by allocating development and test resources to functions on the basis of use. It helps you plan test activities, generate test cases, and select test runs.
Using an operational profile to guide system testing ensures that if testing is terminated and the software is shipped because of imperative schedule constraints, the most-used operations will have received the most testing, and the reliability level will be maximum that is practically achievable for the given test time. In guiding regression testing, it tends to find, among the faults introduced by changes, the ones that have the most effect on reliability.

The three principal terms that we need to understand are the operation, the operation profile, and the operational mode.

### OPERATION

An Operation is a major system logical task of short duration, which returns control to the system when it is complete and whose processing is completely different from other operations.
Major implies that an operation should be related to a functional requirement or feature of a product, not a subtask in the design. The operation is a logical concept, in the sense that it can span a set of software, hardware and human components. It can exist as a series of segments, with a server executing each segment as a process. You can implement the servers on the same or different machines. An operation can be executed in noncontiguous time segments. Short duration implies that there are at least hundreds or thousands of operations executing per hour under normal load conditions. Substantially different processing implies that there is a high probability that an operation is an entity that will contain a fault not found in any other operation. As a rough guideline, we can consider processing as being substantially different if the processing differs from every other operation by at least 100 deliverable executable lines of source code.
Whenever possible, you should define an operation so that it involves processing approximately equal to that required for a natural unit. A user, another system, or the systems own controller can initiate an operation. Some examples of operations are:

1. Command executed by a user (for example, in Fone Follower, phone number entry), sometimes characterized by an input screen that is used to specify the parameters of the command.
2. Response to an input from an external system, such as
   - Processing of a transaction (for example, a purchase, sale, service delivery, or reservation)
   - Processing of an event (for example, an alarm, mechanical movement, or change in state; in the case of Fone Follower, processing a fax call)
3. Routine housekeeping (for example, a security audit, file backup, database audit, or database cleanup) activated by your own system; in the case of Fone Follower, an audit of a section of the phone number database.

### OPERATIONAL PROFILE

The operation profile is simply a set of operations and their probabilities of occurrence. For example, the system operational profile for Fone Follower is shown in the table below. The terms 'pager', 'no pager' indicate that the processing operation is substantially different depending on whether the subscriber has paging service. Similarly, 'answer' and 'no answer' refer to whether the forwardee (the destination where you forward your call) answers; processing is substantially different in these two cases.
A tabular representation of operational profile has a name for each operation, and that name has a probability associated with it.

| Operation | Operations per hour | Probability |
|---|---|---|
| Process voice mail, no pager, answer | 18,000 | 0.18 |
| Process voice mail, no pager, no answer | 17,000 | 0.17 |
| Process voice mail, pager, answer | 17,000 | 0.17 |
| Process fax call | 15,000 | 0.15 |
| ……… | | |
| …… | | |
| …… | | |
| Total | 100,000 | 1 |

A graphical representation presents each operation as a path through a graph, hence the term. The graph consists of a set of nodes, which represent attributes of operations, and branches, which represent different values of the attributes. Each attribute has an associated occurrence probability.

### OPERATIONAL MODE

An operational profile is a distinct pattern of system use and/or set of environmental conditions needing separate testing because it is likely to stimulate different failures. Load test will be divided into operational modes. We will need an operational profile for each operational mode. The same operation will occur in different operational modes, but the occurrence probabilities will be different.
We also need a system operational profile, which consists of the complete set of operations for the system with probabilities of occurrence based on all operational modes.

### PROCEDURE FOR DEVELOPING OPERATIONAL PROFILE

To develop an operational profile, you have to follow the steps listed below:
- Identify the initiators of operations: The initiators of operations include users of the systems, external systems, and the systems own controller.
- Choose between tabular and graphical representation.
- Create an operations list for each (operations) initiator and consolidate results.
- Determine the occurrence rates of the individual operations or attribute values.
- Determine the occurrence probabilities by dividing the individual occurrence rates by the total occurrence rates, of either operations or attribute values, as appropriate.

### STRESS TESTING

When a system is tested with a load, which causes it to allocate its resources in maximum amounts, this is called stress testing. For example, if an operating system is required to handle, 10 interrupts per second and load causes, 20 interrupts per second, the system is being stressed. The goal of stress test is to try to break the system and find the circumstances under which it will crash. This is sometimes called breaking the system.

Stress testing is important because it can reveal defects in real-time and other types of systems, as well as weak areas where poor design could cause unavailability of services. Stress testing often uncovers race conditions, dead locks, depletion of resources in unplanned patterns and upsets in normal operation of the software system. Here system limits and threshold values are exercised, which may not be otherwise revealed under normal testing conditions.

Stress testing is supported by many resources used for performance testing, which can include a load generator also. The testers set the load generator parameters so that load levels cause stress to the system. For example, in a telecommunication system, the arrival rate of calls, the length of the calls, the number of misdials, as well as other system parameters should all be at stress levels. As in the case of performance test, special equipment and laboratory space may be needed for the stress tests.

Stress testing is important from user/client point of view. When system operates correctly under conditions of tests, then clients have confidence that the software can perform as required.

## PERFORMANCE TESTING

An examination of requirements document shows that there are two main types of requirements:
Functional Requirements: Users describe what functions the software should perform. We test for the compliance of these requirements at the system level with the functional-based system tests.
Quality Requirements: These are non-functional in nature but describe quality level expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughputs and delays.

The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test whether there are any hardware or software factors that impact on the systems performance. Is also allows the testers to tune into the system, i.e. optimize the allocation of system resources. For example, testers may find that they need to reallocate memory pools or modify the priority level of certain system operations. It also helps to project the systems future performance levels, which are useful for planning subsequent releases.

Performance testing is designed to test the run-time performance of the software within the integrated system. It occurs throughout all the steps in the testing process. The results of performance tests are quantifiable. At the end of each test, testers will know the performance of the system under test with respect to time. These can be evaluated with respect to requirements objectives.

The Figure 1 below is a representation of examples of special resources needed for a performance test to be carried out.

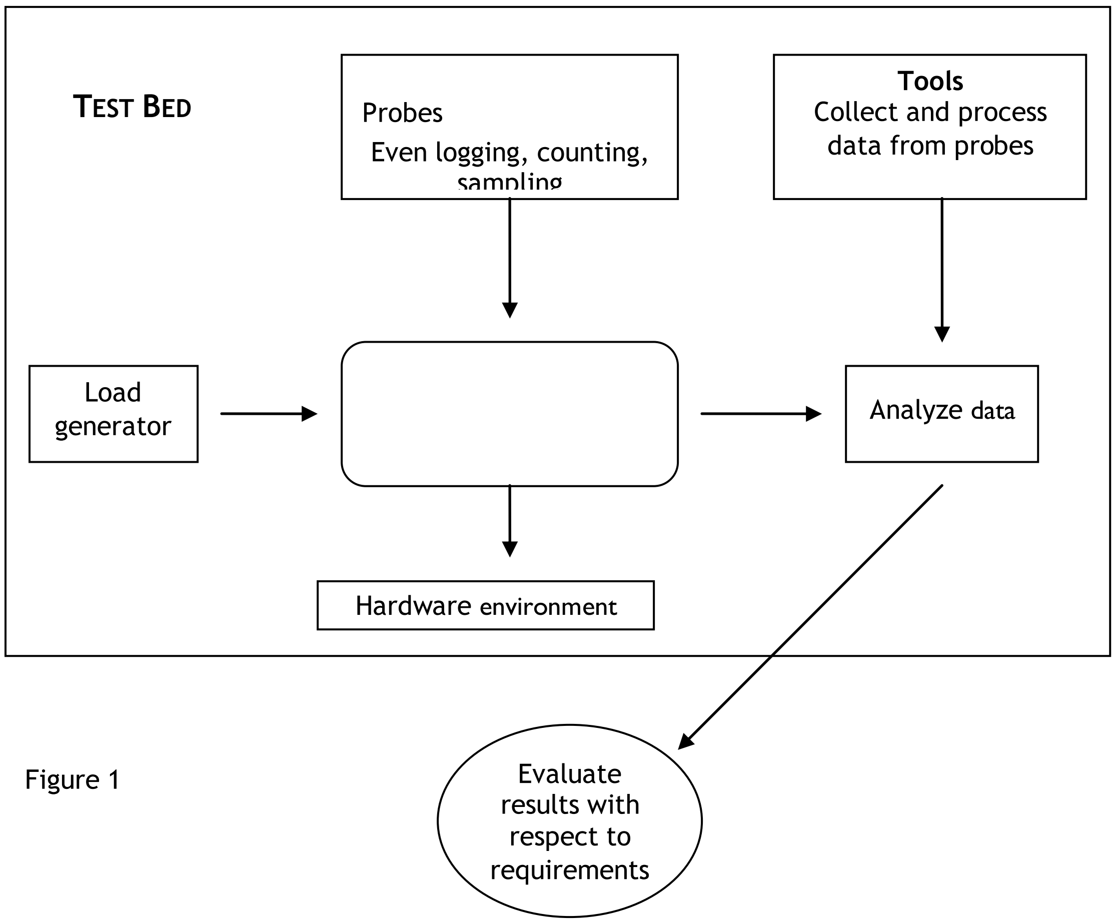

TEST BED
Probes
Even logging, counting, sampling
Tools
Collect and process data from probes
Load generator
Analyze data
Hardware environment
Evaluate results with respect to requirements

Figure 1

Among the resources are:
• A source or transactions to drive the tests, which are represented as a load generator in the figure.
• An experimental test bed that includes hardware and software that the system-under test interacts with. The test bed may sometimes require special laboratory equipment and space reserved for the tests.
• Probes help to collect performance data, which may be hardware or software in nature.
• A set of tools to collect, store, process and interpret the data, which very often collects large volumes of data. Without tools, the testers may have difficulty in processing and analyzing the data in order to evaluate true performance levels.

### SCALABILITY TESTING

Scalability testing is done to make sure that the system under testing will scale with the workload. The system is scalable if it can handle more workload by adding more hardware or external software resources (e.g. switching to an enterprise class database server from desktop class based database server). Scalability testing involves, changing the hardware configuration, software configuration and comparing the load the software can handle against each configuration.

**References:**

PRESSMAN, ROGER S., Software Engineering A Practitioner's Approach, McGraw-Hill, 2001.
Burnstein, ILENE, Practical Software Testing, Springer, 2002.
Musa, John, Software Reliability Engineering, McGraw-Hill 1999.
Lyu, Michael R., Handbook of Software Reliability Engineering, McGraw-Hill, 1996

## POINTS TO PONDER

1. The load generator module of a load-testing tool must be scalable. Comment.

2. Assuming the technology used is good for building for building enterprise systems, does it guarantee that the application will definitely be scalable? Discuss.

3. After performing performance test where we have measured the time taken by an operation multiple times, can we conclude that the test has passed if the average of all the times is less than or equal to the expected time?

4. Why is important to know the point where the system will break by performing stress test?

5. Do you think that load test may be applicable for personal single user applications also e.g. Microsoft Word? Discuss with an example.