Test Case Design Methods

**Learning Objectives**

* Learn the need for systematic Test design
* Learn the different design techniques for doing black box testing and white box testing.
* Introduction to Test Design
* Introduction to Test ware
* Test Case Basics
* Need for Test Design
* Evolution of Test Design Techniques
* Benefits of using Test Design Techniques

**Black Box Test Design Techniques**

* Basic concepts of Black Box Testing
* Equivalence Class Partitioning
* Boundary Value Analysis
* Decision Tables
* State Transition Based Testing
* Orthogonal Arrays
* All Pairs Technique

**White Box Test Design Techniques**

* Basic concepts of White Box Testing
* Types of White Box Testing
* Techniques involved in Static White Box Testing
* Desk Checking
* Code Walkthrough
* Formal Inspections
* Techniques involved in Structural White Box Testing
* Control Flow / Coverage Testing
* Statement Coverage
* Branch Coverage
* Decision / Condition Coverage
* Function Coverage
* Basis Path Testing
* Flow Graph Notation
* Cyclomatic Complexity
* Deriving Test Cases
* Graph Matrices
* Loop Testing
* Simple loops
* Nested loops
* Concatenated loops
* Unstructured loops
* Data Flow Testing
* Practice the concepts learned

What is Testware?

The testware is:

• The subset of software which helps in performing the testing of application.

• Testware are required to plan, design, and execute tests. It contains documents, scripts, inputs, expected results, set-up and additional software or utilities used in testing.

• Testware is term given to combination of all utilities and application software that required for testing a software package.

Testware is special because it has:

* Different purpose.
* Different metrics for quality and.
* Different users.

Testware has its importance through out the lifecycle of the product

Like the software, the testware needs to be saved and maintained.

The role and responsibility of developing the testware lies with the testing team.

Test Case basics

A TEST CASE is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

The process of developing test cases can also help find problems in the requirements or design of an application.

**IEEE Standard 610 (1990) defines a test case as follows:**

“A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.”

A test is not necessarily designed to expose a defect but to gain information. E.g. whether the program will pass or fail the test.

Very often, the information sought involves defects, but not always.

**Test Case Template**

A test case can have the following elements. Note, however, that a test management tool is normally used by companies and the format is determined by the tool used.

* **Test Suite ID** The ID of the test suite to which this test case belongs.

Test Case ID The ID of the test case.

* **Test Case Summary** The summary / objective of the test case.
* **Related Requirement** The ID of the requirement this test case relates/traces to.
* **Prerequisites** Any prerequisites or preconditions that must be fulfilled prior to executing the test.
* **Test Procedure** Step-by-step procedure to execute the test.
* **Test Data** The test data, or links to the test data, that are to be used while conducting the test.
* **Expected Result** The expected result of the test.
* **Actual Result** The actual result of the test; to be filled after executing the test.
* **Status Pass or Fail**. Other statuses can be ‘Not Executed’ if testing is not performed and ‘Blocked’ if testing is blocked.
* **Remarks** Any comments on the test case or test execution.
* **Created By** The name of the author of the test case.
* **Date of Creation** The date of creation of the test case.
* **Executed By** The name of the person who executed the test.
* **Date of Execution** The date of execution of the test.
* **Test Environment** The environment (Hardware/Software/Network) in which the test was executed.

Test Case Example / Test Case Sample

**Test Suite ID** TS001

Test Case ID TC001

**Test Case Summary**

To verify that clicking the Generate Coin button generates coins.

**Related Requirement** RS001

**Prerequisites**

1. User is authorized.
2. Coin balance is available.

**Test Procedure**

1. Select the coin denomination in the Denomination field.
2. Enter the number of coins in the Quantity field.
3. Click Generate Coin.

**Test Data**

1. Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5
2. Quantities: 0, 1, 5, 10, 20

**Expected Result**

1. Coin of the specified denomination should be produced if the specified Quantity is valid (1, 5)
2. A message ‘Please enter a valid quantity between 1 and 10’ should be displayed if the specified quantity is invalid.

**Actual Result**

1. If the specified quantity is valid, the result is as expected.
2. If the specified quantity is invalid, nothing happens; the expected message is not displayed

**Status**

Fail

**Remarks** This is a sample test case.

**Created By** John Doe

**Date of Creation** 01/14/2020

**Executed By** Jane Roe

**Date of Execution** 02/16/2020

**Test Environment**

**OS**: Windows Y

**Browser**: Chrome N

Writing Good Test Cases

* As far as possible, write test cases in such a way that you test only one thing at a time. Do not overlap or complicate test cases. Attempt to make your test cases ‘atomic’.
* Ensure that all positive scenarios AND negative scenarios are covered.

**Language:**

* Write in simple and easy-to-understand language.
* Use active voice instead of passive voice: Do this, do that.
* Use exact and consistent names (of forms, fields, etc).

**Characteristics of a good test case:**

**Accurate:** Exacts the purpose.

**Economical:** No unnecessary steps or words.

**Traceable:** Capable of being traced to requirements.

**Repeatable:** Can be used to perform the test over and over.

**Reusable:** Can be reused if necessary.

Why Test Cases?

* Test cases identify and communicate the conditions that will be implemented in test
* Necessary to verify successful and acceptable implementation of the product requirements
* Help find problems in the requirements or design of an application
* Determine whether we meet the Client expectations
* Help managers make ship / no-ship decisions
* domain is extremely large
* Test case design is required to derive an optimal test suite which is of reasonable size and uncovers as many errors as possible.
* Helps in Arriving an optimal number of test cases.
* A judicious set of tests are designed that test as much of the functionality as possible in a short span of time
* Randomly selecting or writing test cases does not indicate effectiveness of the testing.
* Writing a large number of test cases does not mean that many errors in the system would be uncovered.

Evolution of Techniques

**Phase 1**: Before 1956 – The debugging oriented period – Testing was not separate from debugging

**Phase 2**: Before 1957 - 78 – The demonstration oriented period – Testing was to make sure that the software satisfies its specification

**Phase 3**: Before 1979 - 82 – The destruction oriented period – Testing to detect implementation faults.

**Phase 4**: Before 1983 - 87 – The evaluation oriented period – Testing to detect faults in requirements, design and implementation.

**Phase 5**: After 1988 – The prevention oriented period – Testing to prevent faults in requirements, design and implementation

**Primary objective of designing test cases**

* To have the minimal number of test cases that have the probability of detecting the maximum errors.
* Identify test coverage – a measure of depth and breadth of software being tested
* The depth of coverage indicates the amount of internal logic processing that is tested.
* The breadth of coverage indicates the number of external features and input-output possibilities that are tested.
* To meet this objective, we need to design test cases that meet the following criteria:

**Coverage:**

* The test cases must ensure complete coverage of the software.
* A good test case must allow both valid and invalid data entry to exercise the maximum possible data flow paths in the software.
* The test cases should first focus on individual units and then gradually expand to cover the entire software.

**Modularity:**

Method by which the software is partitioned into similar features and ensured not be tested by more than one test case repeatedly (avoid redundancy).

**Traceability:**

The test cases should be traceable to customer requirements

* Each test design should be uniquely identified.
* Each test design will result in one or more test cases.
* For each test design, specify
* Features to be tested – test items and features
* Approach refinements – application of methods, include rationale for test case selection and packaging of test cases into procedures. Identify method for analyzing test results and tools employed
* Test case identification – list of test cases associated with the design and their objectives
* Feature pass / fail criteria

Black Box Testing Techniques

**Learning Objectives:**

* Understand the basics of Black Box Testing
* Understand the various techniques of Black Box Testing
* Equivalence Class Partitioning
* Boundary Value Analysis
* Decision Tables
* State Transition Based Testing
* Orthogonal Arrays
* All Pairs Technique

Black Box Testing

BLACK BOX TESTING, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional.

This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

* Incorrect or missing functions
* Interface errors
* Errors in data structures or external database access
* Behavior or performance errors
* Initialization and termination errors

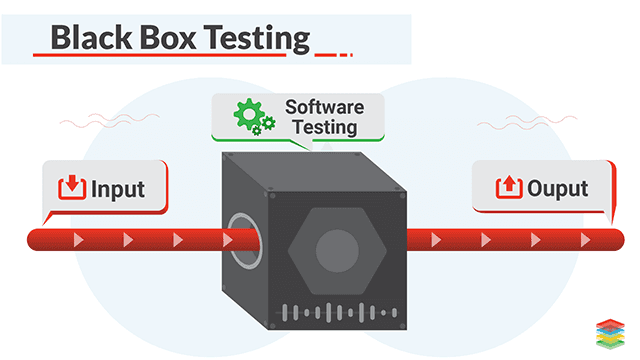
Definition by ISTQB

**black box testing:** Testing, either functional or non-functional, without reference to the internal structure of the component or system.

black box test design technique: Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

**Example**

A tester, without knowledge of the internal structures of a website, tests the web pages by using a browser; providing inputs (clicks, keystrokes) and verifying the outputs against the expected outcome.



Levels Applicable To

Black Box Testing method is applicable to the following levels of software testing:

* Integration Testing
* System Testing
* Acceptance Testing

The higher the level, and hence the bigger and more complex the box, the more black-box testing method comes into use.

Techniques

Following are some techniques that can be used for designing black box tests.

**Equivalence Partitioning:** It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

**Boundary Value Analysis:** It is a software test design technique that involves the determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.

**Cause-Effect Graphing:** It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

Advantages

* Tests are done from a user’s point of view and will help in exposing discrepancies in the specifications.
* Tester need not know programming languages or how the software has been implemented.
* Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
* Test cases can be designed as soon as the specifications are complete.

Disadvantages

* Only a small number of possible inputs can be tested and many program paths will be left untested.
* Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
* Tests can be redundant if the software designer/developer has already run a test case.
* Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.
* Black Box Testing is contrasted with White Box Testing. Read Differences between Black Box Testing and White Box Testing.

What Is Boundary Value Analysis And Equivalence Partitioning?

**Boundary Value Analysis and Equivalence Partitioning explained with a simple example:**

Boundary Value Analysis and Equivalence Partitioning both are test case design strategies in Black-Box Testing.

**Equivalence Partitioning**

In this method, the input domain data is divided into different equivalence data classes. This method is typically used to reduce the total number of test cases to a finite set of testable test cases, still covering maximum requirements.

In short, it is the process of taking all possible test cases and placing them into classes. One test value is picked from each class while testing.

**For Example,** If you are testing for an input box accepting numbers from 1 to 1000 then there is no use in writing thousand test cases for all 1000 valid input numbers plus other test cases for invalid data.

Using the Equivalence Partitioning method above test cases can be divided into three sets of input data called classes. Each test case is representative of a respective class.

So in the above example, we can divide our test cases into three equivalence classes of some valid and invalid inputs.

**Test cases for input box accepting numbers between 1 and 1000 using Equivalence Partitioning:**

**#1)** One input data class with all valid inputs. Pick a single value from range 1 to 1000 as a valid test case. If you select other values between 1 and 1000 the result is going to be the same. So one test case for valid input data should be sufficient.

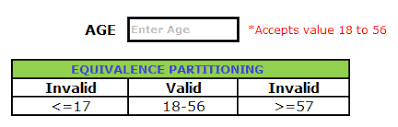
**#2)** Input data class with all values below the lower limit. I.e. any value below 1, as an invalid input data test case.

**#3)** Input data with any value greater than 1000 to represent the third invalid input class.

So using Equivalence Partitioning you have categorized all possible test cases into three classes. Test cases with other values from any class should give you the same result.

We have selected one representative from every input class to design our test cases. Test case values are selected in such a way that largest number of attributes of equivalence class can be exercised.

Equivalence Partitioning uses fewest test cases to cover maximum requirements.



**Boundary Value Analysis**

It's widely recognized that input values at the extreme ends of the input domain cause more errors in the system. More application **errors occur at the boundaries** of the input domain. ‘Boundary Value Analysis' Testing technique is used to identify errors at boundaries rather than finding those that exist in the center of the input domain.

Boundary Value Analysis is the next part of Equivalence Partitioning for designing test cases where test cases are selected at the edges of the equivalence classes.

Test cases for input box accepting numbers between 1 and 1000 using Boundary value analysis:

**#1)** Test cases with test data exactly as the input boundaries of input domain i.e. values 1 and 1000 in our case.

**#2)** Test data with values just below the extreme edges of input domains i.e. values 0 and 999.

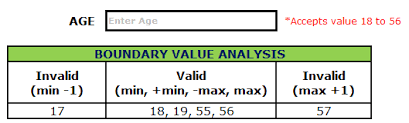
**#3)** Test data with values just above the extreme edges of the input domain i.e. values 2 and 1001.

Boundary Value Analysis is often called as a part of the Stress and Negative Testing.

Note: There is no hard-and-fast rule to test only one value from each equivalence class you created for input domains. You can select multiple valid and invalid values from each equivalence class according to your needs and previous judgments.

**For Example,** if you divided 1 to 1000 input values invalid data equivalence class, then you can select test case values like 1, 11, 100, 950, etc. Same case for other test cases having invalid data classes.

This should be a very basic and simple example to understand the Boundary Value Analysis and Equivalence Partitioning concept.



How To Write Complex Business Logic Test Scenarios Using Decision Table Technique

Decision Table Testing is an easy and confident approach to identify the test scenarios for complex Business Logic.

There are several test case design techniques. In this article, we will learn how to use the Decision Table technique effectively to write test cases for an application with complex Business Logic.

**Here is an illustration:**

We all know that the rules and validations of business take up a major portion of the requirements given by the customers. While observing how these requirements are represented and communicated to the entire project team by Business Analysts or customers, we come to know that most of such business rules and logic and are presented in a logical process flow diagram.

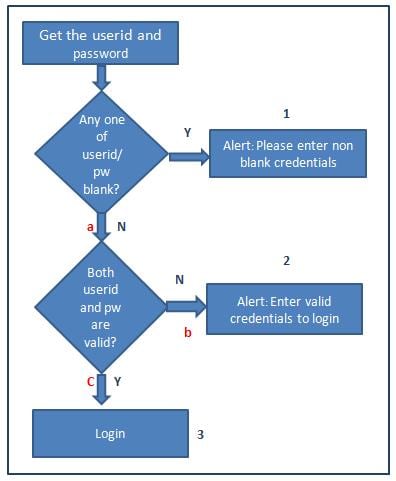
A logical process Flow diagram for a complex requirement comprises of many branches, nodes, and decision boxes. Hopefully, we testers are expected to cover all those branches and touch every nook and corner of such a complex logical tree. I have also faced such complex business flows and tried many test case/test scenario preparation techniques for making the process easier.

Finally, I found the Decision Table Testing technique to be highly useful in this aspect. Here is how a Decision Table technique can make the test scenario preparation for complex Business Logic easier.

**Example: Writing Test cases for a login screen using the Decision Table Technique:**

Let’s take a Decision Table example of below business requirement for a login screen.

Fig: 1.0 Sample business flow diagram



The first step we do is to name all the branches and leave with numbers or alphabets as below.

1, 2, 3 are the leaves and a, b & c are the branches.

Then, we have to create a Decision table as shown below:

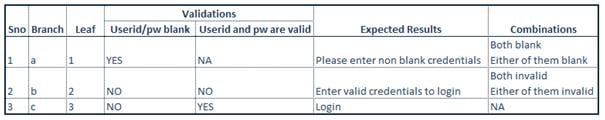


Fig 1.1 Decision table for business flow fig 1.0

**Points to Remember**

* All the validations specified in the decision boxes should be made of the columns on the table.
* All the results (leaves) mentioned in the flow diagram should be covered in the decision table.
* All combinations of inputs needed to obtain a certain result shall be mentioned in the combinations column and can be included while writing the test cases.

After completing the Decision table, one has to just verify whether all the branches and leaves in the logical tree are covered.

**Advantages Of Using Decision Table Technique**

**#1)** Any complex Business flow represented as a diagram can be easily covered in this technique.

**#2)** It provides quick confidence on the test cases. One need not have to review his own test cases multiple times to gain confidence.

**#3)** Easy to understand. Anyone can make test cases from this Decision table template.

**#4)** Rework on the test cases and test scenarios can be totally avoided, as it gives complete coverage at the first shot.

**Limitations Of Using Decision Table Technique**

**#1)** Certain test case preparation techniques like Boundary value analysis, Equivalence partitioning cannot be directly accommodated in this template. But, one can note it down in the combinations column and use them while writing test cases.

Before explaining why other test case writing techniques cannot assure as much accuracy as Decision tables, I would like to quickly remind other Black box and White box test case writing techniques.

**Other Test Case Design Techniques**

**#1)** Boundary Value Analysis is a Software Testing technique in which test cases are designed to include representatives of boundary values in and out of a given range.

**#2)** Equivalence Partitioning also, called Equivalence Class Partitioning is a Software Testing technique that divides the given condition into partitions and one input data from each partition can be chosen for testing.

**#3)** State Transition testing is a black-box testing technique, which can be used to design test cases for a system that acquires a finite number of states and can transit from one state to another upon specific events.

**#4)** Error Guessing is a technique where the experience of a tester is used to find the errors or part of an application with the highest possibility of finding errors. This is a skill-based technique without any rules.

**#5)** Use Case testing In this technique, use cases/scenarios are used to write the test cases. The interaction of users and systems is described in a use case.

**Some more Test Design techniques:**

#6) Statement coverage

#7) Condition Coverage

#8) Exploratory testing

**Why can't other test case design techniques for Business logic prove to be useful as Decision Tables?**

**#1)** Boundary Value Analysis and Equivalence class partitioning is meant for numeric ranges and length. Both of these techniques alone cannot ensure 100% Test Coverage for business rules.

**#2)** Error Guessing is more about the experience. Though experience is required, it cannot prove to be everything.

**#3)** With the State Transition testing technique, one can ensure that all parts of the logical tree are covered but it does not suggest document or artifact as Decision Table technique ensures coverage with a decision table (fig 1.1).

**Conclusion**

For writing test cases for business logic, it is advisable to follow the below steps to prepare test cases so as to ensure maximum Test Coverage:

**Step #1)** Use a Decision Table test case design technique to attain 100% logical coverage.

**Step #2)** Boundary Value Analysis and Equivalence partitioning for covering various ranges of inputs.

**Step #3)** Combinations and permutations for field level validations (though not all permutations are required).

**Step #4)** Error guessing (apart from the errors that can be identified from the above three steps) with experience as a final touch

**State Transition Testing Technique And State Transition Diagram With Examples**

**State Transition Testing**

State Transition testing is a Black-box testing technique, which can be applied to test ‘Finite State Machines’.

A ‘Finite State Machine (FSM)’ is a system that will be in different discrete states (**like “ready”, “not ready”, “open”, “closed”,…**) depending on the inputs or stimuli.

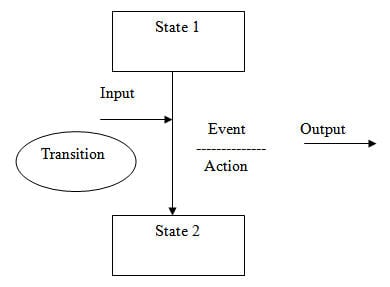
The discrete states that the system ends up with, depends on the rules of the transition of the system. That is, if a system gives a different output for the same input, depending on its earlier state, then it is a finite state system.

Further, if every transaction is tested in the system, it is called “0-switch” coverage. If testing covers 2 pairs of valid transactions, then it is “1-switch” coverage, and so on.

**What Is The State Transition Testing Technique?**

State transition technique is a dynamic testing technique, which is used when the system is defined in terms of a finite number of states and the transitions between the states are governed by the rules of the system.

Or in other words, this technique is used when features of a system are represented as states which transform into one another. The transformations are determined by the rules of the software. The pictorial representation can be shown as:



So here we see that an entity transitions from State 1 to State 2 because of some input condition, which leads to an event and results in action and finally gives the output.

**To explain it with an example:**

You visit an ATM and withdraw $1000. You get your cash. Now you run out of balance and make exactly the same request of withdrawing $1000. This time ATM refuses to give you the money because of insufficient balance. So, here the transition, which caused the change in state is the earlier withdrawal

**State Transition Testing Definition**

Having understood what State Transition is, we can now arrive at a more meaningful definition for State Transition testing. So, it is a kind of black-box testing in which the tester has to examine the behavior of AUT (Application Under Test) against various input conditions given in a sequence.

The behavior of the system is recorded for both positive and negative test values.

**When to use State Transition Testing?**

State Transition testing can be employed in the following situations:

* When the application under test is a real-time system with different states and transitions encompassed.
* When the application is dependent upon the event/values/conditions of the past.
* When the sequence of events needs to be tested.
* When the application needs to be tested against a finite set of input values.

When not to use State Transition Testing?

You should not rely upon State Transition testing under the following situations:

When testing is not required for sequential input combinations.

When different functionalities of the application are required to be tested (more like Exploratory testing).

State Transition Testing Example in Software Testing

In the practical scenario, testers are normally given the State Transition diagrams and we are required to interpret it.

These diagrams are either given by the Business Analysts or a stakeholder and we use these diagrams to determine our test cases.

**Let’s consider the below situation:**

* Software name – Manage\_display\_changes
* Specifications – The software responds to input requests to change display mode for a time display device.

The Display mode can be set to one of the four values:

* Two corresponding to displaying either the time or date.
* The other two when altering either the time or the date.

The different states are as follows:

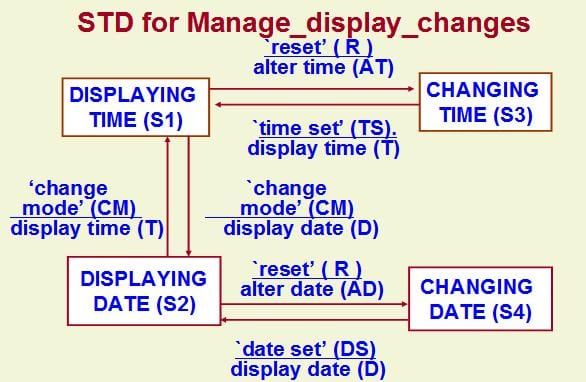
**Change Mode (CM):** Activation of this shall cause the display mode to move between “display time (T)” and “display date (D)”.

**Reset (R):** If the display mode is set to T or D, then a “reset” shall cause the display mode to be set to “alter time (AT)” or “alter date (AD)” modes.

**Time Set (TS):** Activation of this shall cause the display mode to return to T from AT.

**Date Set (DS):** Activation of this shall cause the display mode to return to D from AD.

**State Transition diagram**



Now, let’s move to interpret it:

Here:

**#1) Various States are:**

Display Time(S1),

Change Time(S3),

Display Date(S2), and

Change Date (S4).

**#2) Various Inputs are:**

Change Mode(CM),

Reset (R),

Time Set(TS),

Date Set(DS).

**#3) Various Outputs are:**

Alter Time(AT),

Display Time(T),

Display Date(D),

Alter Date (AD).

**#4) The Changed States are:**

Display Time(S1),

Change Time (S3),

Display Date (S2) and

Change Date (S4).

**Step 1:** Write all of the start states. For this, take one state at a time and see how many arrows are coming out from it.

For State S1, there are two arrows coming out of it. One arrow is going to state S3 and another arrow is going to state S2.

For State S2 – There are 2 arrows. One is going to State S1 and other going to S4

For State S3 – Only 1 arrow is coming out of it, going to state S1

For State S4 – Only 1 arrow is coming out of it, going to state S2

Let’s put this on our table:



Since for state S1 and S2, there are two arrows coming out, we have written it twice.

Step -2: For each state, write down their final transitioned states.

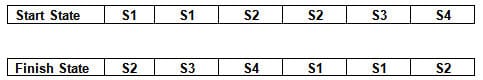
For state S1 – The final states are S2 and S3

For State S2 – The final states are S1 and S4

For State S3 – The final state is S1

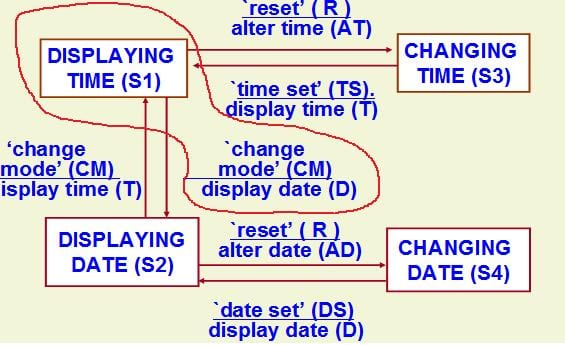
For State S4 – Final State is S2

Put this on the table as an Output/Resultant state.

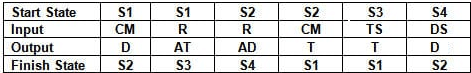


Step 3: For each start state and its corresponding finish state, write down the input and output conditions

– For state S1 to go to state S2, the input is Change Mode (CM) and output is Display Date(D) shown below:

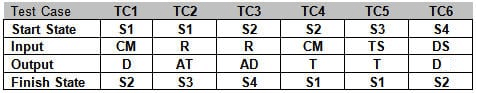


In a similar way, write down the Input conditions and its output for all the states as follows:

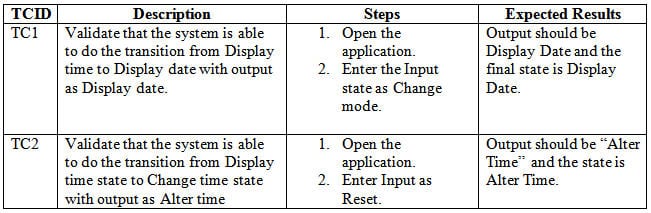


Step 4:

Now add the test case ID for each test shown below:



Now let’s convert it to formal test cases:



In this way, all the remaining test cases can be derived. I assume the other attributes of the test cases like preconditions, severity, priority, environment, build, etc. are also included in the test case.

**Summarizing the steps once again**:

Identify the initial states and their final state based on the lines/arrows that are coming out of the initial state.

For each initial state, find out the input condition and the output result

Mark each set as a separate test case.

What Is Orthogonal Array Testing (OATS)?

Orthogonal Array Testing technique is a statistical approach for testing pair-wise interactions. Most of the defects which I have observed are caused due to interaction and integration.

This interaction or integration can be within different objects, elements, options in a screen of the application, or configuration setting in a file. Such a combination of objects and elements results in the functioning of the application.

It is obvious that some of the combinations are missed to test which thereby results in insufficient tests. Hence in order to cover the entire functionality in the testing scope with the correct amount of combinations to be tested, Orthogonal Array Testing is used.

This is a Combinational Test Technique that ensures that the complete functionality of an application is tested with a limited and proportionate amount of combinations under test without compromising on the quality of testing.

The beauty of this technique is that it maximizes the coverage by a comparatively lesser number of test cases. The pairs of parameters which are identified should be independent of each other. It’s a black box technique, so like other BB techniques; we don’t need to have the implementation knowledge of the system. The point here is to identify the correct pair of input parameters.

There are many techniques of CTD, where the OATS (Orthogonal Array Testing Technique) is widely used.

**Terminologies in Orthogonal Array Testing**

Before understanding the actual implementation of Orthogonal Array Testing, it is essential to understand the terminologies related to it.

Enlisted below are the widely used terminologies in Orthogonal Array Testing:

**Term Description**

**Runs** It is the number of rows which represents the number of test conditions to be performed.

**Factors** It is the number of columns which represents in the number of variable to be tested

**Levels**  It represents the number of values for a Factor

As the rows represent the number of test conditions (experiment test) to be performed, the goal is to minimize the number of rows as much as possible.

Factors indicate the number of columns which is the number of variables.

Levels represent the maximum number of values for a factor (0 – levels – 1). Together, the values in Levels and Factors are called LRUNS (Levels\*\*Factors).

**Implementation Technique Of OATS**

The Orthogonal Array Testing technique has the following steps:

**#1)** Decide the number of variables that will be tested for interaction. Map these variables to the factors of the array.

**#2)** Decide the maximum number of values that each independent variable will have. Map these values to the levels of the array.

**#3)** Find a suitable orthogonal array with the smallest number of runs. The number of runs can be derived from various websites. One such website is listed here.

**#4)** Map the factors and levels onto the array.

**#5)** Translate them into the suitable Test Cases

**#6)** Look out for the leftover or special Test Cases (if any)

After performing the above steps, your Array will be ready for testing with all the possible combinations covered.

**Example**

We provide our personal information like Name, Age, qualification, etc., in various registration forms like first-time app installation or any other Government websites.

The following example is from such kind of application form. Consider that there are four fields in a registration form (webpage) which are having certain sub-options in it.

Age field

* Less than 18
* More than 18
* More than 60

Gender field

* Male
* Female
* NA

Highest Qualification

* High School
* Graduation
* Post-Graduation

Mother Tongue

* Hindi
* English
* Other

**Step 1:** Determine the number of independent variables. There are four independent variables (Fields of the registration form) = 4 Factors.

**Step 2:** Determine the maximum number of values for each variable. There are three values (There are three sub-options under each field) = 3 Levels.

**Step 3:** Determine the Orthogonal Array with 4 Factors and 3 Levels. Referring to the link we have derived the number of rows required i.e. 9 Rows.

Orthogonal array follows the pattern LRuns(LevelsFactors). Hence in this example, the Orthogonal Array will be L9(34).

Thus the Orthogonal Array will look as given below.

| **Runs** | **Factor 1** | **Factor 2** | **Factor 3** | **Factor 4** |
| --- | --- | --- | --- | --- |
| Run 1 | 0 | 0 | 0 | 0 |
| Run 2 | 0 | 1 | 2 | 1 |
| Run 3 | 0 | 2 | 1 | 2 |
| Run 4 | 1 | 0 | 2 | 2 |
| Run 5 | 1 | 1 | 1 | 0 |
| Run 6 | 1 | 2 | 0 | 1 |
| Run 7 | 2 | 0 | 1 | 1 |
| Run 8 | 2 | 1 | 0 | 2 |
| Run 9 | 2 | 2 | 2 | 0 |

Step no. 4: Map the Factors and Levels of the Array generated.

“Factor 1” will be replaced by AGE.

“Factor 2” will be replaced by Gender.

“Factor 3” will be replaced by Highest Qualification.

“Factor 4” will be replaced by Mother Tongue.

0, 1, 2 will be replaced by each sub-option under their respective Factor (field).

After mapping the Factors and Levels, the Orthogonal Array will look as shown below:

| **Runs** | **AGE** | **Gender** | **Highest Qualification** | **Mother Tongue** |
| --- | --- | --- | --- | --- |
| Run 1 | Less than 18 | Male | High School | Hindi |
| Run 2 | Less than 18 | Female | Post-Graduation | English |
| Run 3 | Less than 18 | NA | Graduation | Other |
| Run 4 | More than 18 | Male | Post-Graduation | Other |
| Run 5 | More than 18 | Female | Graduation | Hindi |
| Run 6 | More than 18 | NA | High School | English |
| Run 7 | More than 60 | Male | Graduation | English |
| Run 8 | More than 60 | Female | High School | Other |
| Run 9 | More than 60 | NA | Post-Graduation | Hindi |

Step no. 5: Each Run in the above table represents the test scenario to be covered in testing. Each run is changed to a test condition.

**Advantages Of Orthogonal Array Testing**

This technique is beneficial when we have to test with a huge number of data having many permutations and combinations.

* Less number of Test conditions which requires less implementation time.
* Less Execution time.
* Easy Analysis of Test condition due to less number of Test conditions.
* High coverage of codes.
* Increased overall productivity and ensures that the quality test is performed.

**Limitations Of OATS**

None of the testing technique provides a guarantee of 100% coverage. Each technique has its way of selecting the test conditions. On similar lines, there are

**some limitations to using this technique:**

* Testing will fail if we fail to identify the good pairs.
* Probability of not identifying the most important combination which can result in losing a defect.
* This technique will fail if we do not know the interactions between the pairs.
* Applying only this technique will not ensure complete coverage.
* It can find only those defects which arise due to pairs, as input parameters.

**Conclusion**

Orthogonal Array testing is a systematical and statistical way of testing pair-wise interactions. It is done by deriving small sets of test cases from a large number of scenarios and also by giving precedence to factors and levels that appear multiple times in the combinatorial outputs.

**We can use Orthogonal Array testing in our day to day application testing by:**

* Forming systematic, statistical pair-wise combinations of factors across their levels.
* Creating an optimized test suite with fewer test scenarios and generating negative test case optimization.
* Detecting all single, double, and triple mode defects in the given input combinations.
* Executing a concise set of tests and uncovering most of the bugs.
* Now as you have a clear understanding of the implementation of Orthogonal Array testing, you can easily implement it in your application or webpage which will cover all the aspects of the functionality of the application in a limited number of test cases.

All-pairs Testing

All-pairs testing technique is also known as pairwise testing. It is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input (radio button is used when you have to select only one option for example when you select gender male or female, you can select only one option), list box, text box, etc.

Suppose, you have a function of a software application for testing, in which there are 10 fields to input the data, so the total number of discrete combinations is 10 ^ 10 (100 billion), but testing of all combinations is complicated because it will take a lot of time.

So, let's understand the testing process with an example:

Assume that there is a function with a list box that contains 10 elements, text box that can accept 1 to 100 characters, radio button, checkbox and OK button.

The input values are given below that can be accepted by the fields of the given function.

* Check Box - Checked or Unchecked
* List Box - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
* Radio Button - On or Off
* Text Box - Number of alphabets between 1 to 100.
* OK - Does not accept any value, only redirects to the next page.

Calculation of all the possible combinations:

Check Box = 2

List Box = 10

Radio Button = 2

Text Box = 100

Total number of test cases = 2\*10\*2\*100

= 4000

The total number of test cases, including negative test cases, is 4000.

Testing of 4000 positive and negative test cases, is a very long and time-consuming process. Therefore, the task of the testing team is to reduce the number of test cases, to do this, the testing team considers the list box values in such a way that the first value is 0, and the other value can be any numeric number, neither positive nor negative. Ten values are now converted into 2 values.

Values of checkbox and radio button cannot be reduced because each has a combination of only 2 values. At last, the value of the text box is divided into three input categories valid integer, invalid integer, and alpha-special character.

Now, we have only 24 test cases, including negative test cases.

2\*2\*2\*3 = 24

Now, the task is to make combinations for all pair technique, into which each column should have an equal number of values, and the total value should be equal to 24.

In order to make text box column, put the most common input on the first place that is a valid integer, on the second place put the second most common input that is an invalid integer, and at the last place put the least common input that is an Alpha Special Character.

Then start filling the table, the first column is a text box with three values, the next column is a list box that has 2 values, the third column is a checkbox that has 2 values, and the last one is a radio button that also has 2 values.

In order to make text box column, put the most common input on the first place that is a **valid integer**, on the second place put the second most common input that is an **invalid integer**, and at the last place put the least common input that is an **Alpha Special Character.**

Then start filling the table, the first column is a text box with three values, the next column is a list box that has 2 values, the third column is a checkbox that has 2 values, and the last one is a radio button that also has 2 values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Text box** | **List Box** | **Check Box** | **Radio Button** |
| Valid Integer | 0 | Check | ON |
| Invalid Integer | Other | Uncheck | OFF |
| Valid Integer | 0 | Check | ON |
| Invalid Integer | Other | Uncheck | OFF |
| AlphaSpecialCharacter | 0 | Check | ON |
| AlphaSpecialCharacter | Other | Uncheck | OFF |

In the table, we can see that the conventional software method results in 24 test cases instead of 4000 cases, and the pairwise testing method only in just 6 pair test cases.

**Cause and Effect Graph in Black box Testing**

Cause-effect graph comes under the black box testing technique which underlines the relationship between a given result and all the factors affecting the result. It is used to write dynamic test cases.

The dynamic test cases are used when code works dynamically based on user input. For example, while using email account, on entering valid email, the system accepts it but, when you enter invalid email, it throws an error message. In this technique, the input conditions are assigned with causes and the result of these input conditions with effects.

Cause-Effect graph technique is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of the software.

# **Cause and Effect Graph in Black box Testing**

Cause-effect graph comes under the black box testing technique which underlines the relationship between a given result and all the factors affecting the result. It is used to write dynamic test cases.

The dynamic test cases are used when code works dynamically based on user input. For example, while using email account, on entering valid email, the system accepts it but, when you enter invalid email, it throws an error message. In this technique, the input conditions are assigned with causes and the result of these input conditions with effects.

Cause-Effect graph technique is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of the software.

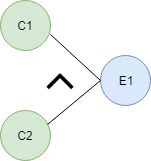
The main advantage of cause-effect graph testing is, it reduces the time of test execution and cost.

This technique aims to reduce the number of test cases but still covers all necessary test cases with maximum coverage to achieve the desired application quality.

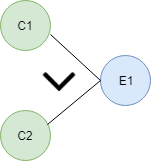
Cause-Effect graph technique converts the requirements specification into a logical relationship between the input and output conditions by using logical operators like AND, OR and NOT.

## **Notations used in the Cause-Effect Graph**

**AND -** E1 is an effect and C1 and C2 are the causes. If both C1 and C2 are true, then effect E1 will be true.



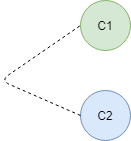
**OR -** If any cause from C1 and C2 is true, then effect E1 will be true.



**NOT -** If cause C1 is false, then effect E1 will be true.

Cause and Effect Graph in Black box testing

**Mutually Exclusive -** When only one cause is true.

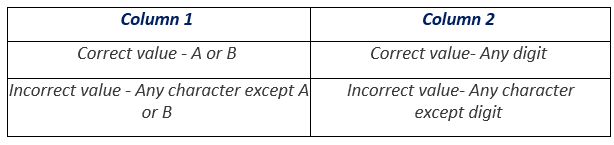


**Let's try to understand this technique with some examples:**

### **Situation:**

The character in column 1 should be either A or B and in the column 2 should be a digit. If both columns contain appropriate values then update is made. If the input of column 1 is incorrect, i.e. neither A nor B, then message X will be displayed. If the input in column 2 is incorrect, i.e. input is not a digit, then message Y will be displayed.

* A file must be updated, if the character in the first column is either "A" or "B" and in the second column it should be a digit.
* If the value in the first column is incorrect (the character is neither A nor B) then massage X will be displayed.
* If the value in the second column is incorrect (the character is not a digit) then massage Y will be displayed.



Now, we are going to make a Cause-Effect graph for the above situation:

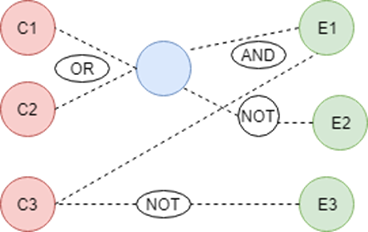
**Causes are:**

* C1 - Character in column 1 is A
* C2 - Character in column 1 is B
* C3 - Character in column 2 is digit!

**Effects:**

* E1 - Update made (C1 OR C2) AND C3
* E2 - Displays Massage X (NOT C1 AND NOT C2)
* E3 - Displays Massage Y (NOT C3)

Where AND, OR, NOT are the logical gates.



**Effect E1- Update made-** The logic for the existence of effect E1 is "**(C1 OR C2) AND C3**". For **C1 OR C2**, any one from C1 and C2 should be true. For logic **AND C3** (Character in column 2 should be a digit), C3 must be true. In other words, for the existence of effect E1 (Update made) any one from C1 and C2 but the C3 must be true. We can see in graph cause C1 and C2 are connected through OR logic and effect E1 is connected with AND logic.

**Effect E2 - Displays Massage X -** The logic for the existence of effect E2 is "**NOT C1 AND NOT C2**" that means both C1 (Character in column 1 should be A) and C2 (Character in column 1 should be B) should be false. In other words, for the existence of effect E2 the character in column 1 should not be either A or B. We can see in the graph, **C1 OR C2** is connected through NOT logic with effect E2.

**Effect E3 - Displays Massage Y-** The logic for the existence of effect E3 is "**NOT C3**" that means cause C3 (Character in column 2 is a digit) should be false. In other words, for the existence of effect E3, the character in column 2 should not be a digit. We can see in the graph, **C3** is connected through NOT logic with effect E3.

So, it is the cause-effect graph for the given situation. A tester needs to convert causes and effects into logical statements and then design cause-effect graph. If function gives output (effect) according to the input (cause) so, it is considered as defect free, and if not doing so, then it is sent to the development team for the correction.

### **Conclusion**

Summary of the steps:

* Draw the circles for effects and Causes.
* Start from effect and then pick up what is the cause of this effect.
* Draw mutually exclusive causes (exclusive causes which are directly connected via one effect and one cause) at last.
* Use logic gates to draw dynamic test cases.

White Box Testing Techniques

WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.

This method is named so because the software program, in the eyes of the tester, is like a white/transparent box; inside which one clearly sees.

**Definition by ISTQB**

white-box testing: Testing based on an analysis of the internal structure of the component or system.

white-box test design technique: Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

**Example**

A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.

White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.

**Levels Applicable To**

White Box Testing method is applicable to the following levels of software testing:

* Unit Testing: For testing paths within a unit.
* Integration Testing: For testing paths between units.
* System Testing: For testing paths between subsystems.

However, it is mainly applied to Unit Testing.

**Advantages**

Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.

Testing is more thorough, with the possibility of covering most paths.

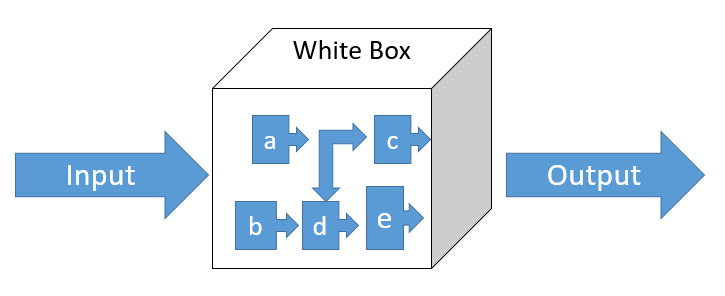
Disadvantages

Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.

Test script maintenance can be a burden if the implementation changes too frequently.

Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

WhiteBox testing Example



**White Box Testing – Why?**

Especially useful for:

Detecting special case logical errors

Uncovering incorrect programming assumptions

Debugging code

Finding random typographical errors

**White Box Testing – When?**

In the Software testing life cycle, White Box Testing is used once the structural details (low level design, implementable code), of the application are available.

White box testing can be applied at all levels of system development – unit, integration and system.

White box is generally more equated and associated with unit testing by developers, however, these techniques can be used for other developmental artifacts like requirements, designs and test cases.

White Box Testing - Guidelines

**Applicability**

White box testing can be applied at all levels of system development

unit, integration, and system.

White box testing is more than code testing—it is path testing.

The same techniques can be applied to test paths between modules within subsystems, between subsystems within systems, and even between entire systems.

**Advantages**

White-box testing enables a tester to execute:

All the independent paths within a module at least once

All logical decisions on their true and false sides

All loops at their boundaries and within their operational bounds

Internal data structures to ensure their validity

**Limitations**

The number of execution paths may be so large then they cannot all be tested.

The test cases chosen may not detect data sensitivity errors.

For example: p=q/r; may execute correctly except when r=0.

White box testing 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 testing.

The tester must have the programming skills to understand and evaluate the software under test.

**White Box Testing - Techniques**

Techniques commonly used

* Static White Box Testing
* Desk Checking
* Code Walkthrough
* Formal Inspections

**Structural White Box Testing**

* Control Flow / Coverage Testing
* Basis Path Testing
* Loop Testing
* Data Flow Testing

**White Box Testing – Static Testing**

Static Testing:

* Type of testing which involves only the source code of the product and not the binaries or executables.
* It is done before the code is completed or executed
* It involves selected people going through the code to find out the defects

Primary aim of Static Testing is to see if

* Code is according to the Functional Requirements (FS)
* Code is according to the design
* Code adheres to coding standards
* All functionalities covered in code
* Error handling

Types of Static Testing Techniques

* Desk Checking
* Code Walkthrough's
* Formal Inspections or Fagan Inspection

**Static Testing - Desk Checking**

Introduction

* First kind of testing done on the code
* Done by the developer of the code
* Code is compared with the design or specifications to make sure that it is in accordance with the specified details
* Done before the code is compiled or executed
* Errors are detected and corrected by the author of the code
* No structured or formal method for completeness
* No maintenance of log or checklist

Desk Checking – Advantages

* Done by the author of the code who knows about the code and the programming language very well
* Can be done quickly without much dependency on other developers or testers.
* Defects detected at this phase are easy to locate and correct
* Defects are corrected without any delay

Desk Checking – Disadvantages

* Author of the code may not have the mentality of detecting errors or problems with his own code!
* Developers prefer to write new code rather than to do any form of testing!
* Relies heavily on the author (person dependency is high)

**Static Testing - Code Walkthrough**

**Technique**

* Also called as Technical Code Walkthrough
* A group of technical people (including the author) go through the code
* It is a semi-formal review
* Generally the people involved in the code walkthrough reviews are technical leads, database administrators and one or more peers.
* The group of people go through the code and raise questions for the author
* The author explains the logic and if there is any mistake, the author notes it down and corrects the same

**Code Walkthrough – Advantages**

* Since a group of technical people look at the code, defects related to DB, code logic etc are easily detected
* Ensures that coding standards are met

**Code Walkthrough – Disadvantages**

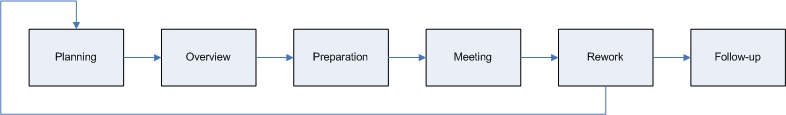
Depends on the availability of the group, the review process may get delayed

It may not be possible to review 100% of the code in this method as the panel may not have the bandwidth to accommodate the same.

**Static Testing – Formal Inspections**

**Technique**

* Also called as Fagan Inspection (FI) (named after Micheal Fagan who introduced the concept of software inspections)
* It is a structured process of finding defects in the source code
* It is a formal review aimed at detecting all faults, violations and other side effects.
* FI takes place only after the author has completed desk checking and walkthroughs.
* The review process comprises of the following phases – Planning, Preparation, Inspection Meeting, Rework and Follow up



**Static Testing – Formal Inspections**

Inspection Meeting involves the process in which the code is inspected and defects are found. Defects are noted down and handed to the author

Rework is done to fix the defect. The code is corrected by the author.

Follow up is done by the moderator to ensure that the defect has been fixed correctly.

Generally a checklist is used for the formal reviews

**Formal Inspections – Advantages**

* Inspections are planned and systematically carried out which ensures higher efficiency and effectiveness as compared to other techniques
* Increased review time ensures larger coverage of artifact and uncovering more errors
* Criticality of artifacts like requirements and designs require formal techniques to ensure correctness, which is provided by inspections

**Formal Inspections – Disadvantages**

* Planning becomes difficult due to the availability of participants and this may cause delay in carrying out inspections
* Requires more time and effort, hence costs more
* Requires higher skill sets which may prove to be a limitation

**White Box Testing – Structural Testing**

**Structural Testing:**

Testing takes into account the code, code structure, internal design and how they are coded.

Commonly used techniques for structural testing are

* Control Flow / Coverage Testing
* Statement Coverage
* Branch Coverage
* Decision / Condition Coverage
* Function Coverage
* Basis Path Testing
* Flow Graph Notation
* Cyclomatic Complexity
* Deriving Test Cases
* Graph Matrices
* Loop Testing
* Simple loops
* Nested loops
* Concatenated loops
* Unstructured loops
* Data Flow Testing

**Working process of white box testing:**

Input: Requirements, Functional specifications, design documents, source code.

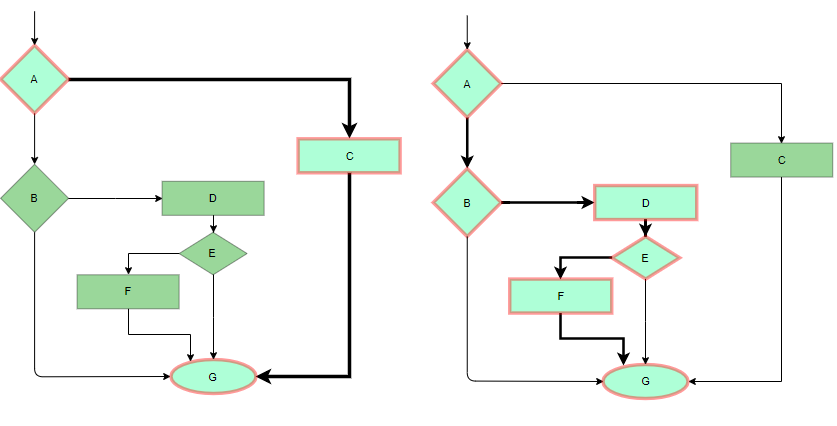
Processing: Performing risk analysis for guiding through the entire process.

Proper test planning: Designing test cases so as to cover entire code. Execute rinse-repeat until error-free software is reached. Also, the results are communicated.

Output: Preparing final report of the entire testing process.

**Testing techniques:**

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 comprises of testing of the following

* Sequential Control Flow
* Two way decision – IF THEN ELSE statements
* Multi-way decision – SWITCH statements
* Loops – DO, DO WHILE, UNTIL, FOR statements

**Technique**

* Write test cases that execute each of the program statements
* Each of the above mentioned statements that are present in the program need to be covered
* For the section of program that comprises of a ‘sequential control flow’ write test cases that execute from top to bottom (without any branches)
* For two-way decision –IF THEN ELSE statements ensure that there is at least one test case that specifically covers each of the IF and ELSE portions
* For multi-way switch statements, separate test cases need to be present to test each value in the switch statement
* A huge percentage of defects in a system are attributed due to issues with the loop statement – FOR, WHILE, DO WHILE etc
* Ensure tests are written such that
* boundary conditions of the loop are covered
* execute the loop between minimum and maximum possible conditions and check all ‘normal’ working values of the loop
* Test the execution when the conditions for the loop are not satisfied (ie when the loop is skipped)

The amount of test coverage for a program can be got by

Statement Coverage = Total Statements Executed / Total no of executable statements in the program X 100

Advantage of Statement Coverage Testing

* Gives an indication about the percentage of statements executed in testing
* It is an exhaustive method of testing where testing aims at covering all statements within the program

Disadvantages of Statement Coverage Testing

* Since this technique involves in testing all the statements within the program, it would lead to an exhaustive set of test cases which may be impossible in practical purposes.

Sometimes the percentage of coverage may not give a right picture.

Let Value = 0;

If (A == B) {

statement 1;

statement 1;

statement 1;

statement 1;

statement 1;

}

Else C = A+B/Value; (Divide by 0 error!)

Here since the “IF” portion works fine, we get a code coverage of 87%.

But if the program is written in such a way that A will not be equal to B 90% of the times, then, then we are left with a defect that hits the user 90% of the time

**Branch Coverge:** 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.

Condition coverage is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome

Ensure that all conditions involved in the program are tested

Test Cases are designed such that each component of composite condition expression are tested for both TRUE and FALSE conditions ie ALL outcomes (TRUE and FALSE) of each condition are tested

Condition Coverage is defined by the following formula:

**Condition Coverage = (Total Decisions Exercised / Total Number of Decisions in the program ) \* 100**

**Example**

Consider the conditional Expression:

IF ((A<20 AND B>10) OR C == 5)

In Condition Coverage Testing each of the values A, B and C are exercised atleast once for both TRUE and FALSE values

(NOTE: In Path Testing, we would have given one just condition eg. C==5 or A =15 and B= 15 to test the IF-THEN portion. We would have not tested for all combinations of the condition)

**Multiple Condition Coverage:** In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let’s consider the following example:

READ X, Y

IF(X == 0 || Y == 0)

PRINT ‘0’

#TC1: X = 0, Y = 0

#TC2: X = 0, Y = 5

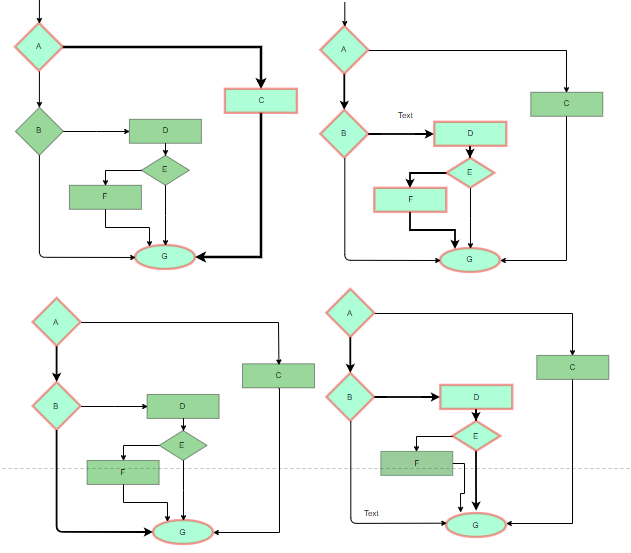
#TC3: X = 55, Y = 0

#TC4: X = 55, Y = 5

Hence, four test cases required for two individual conditions.

Similarly, if there are n conditions then 2n test cases would be required.

**Branch Coverge:** 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

**Control Flow Testing – Function Coverage**

**Technique**

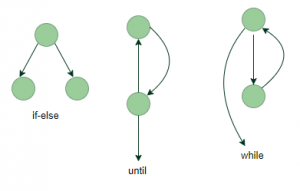
* Most programs are realized by calling a set of functions.
* Requirements of a product are mapped to functions during the design phase
* Each function is the smallest logical unit that does a specific functionality.
* There could be functions for computing the average of 10 numbers, inserting a row into the database, calculating the premium etc.
* Tests are written to exercise each of the different functions in the code

**Advantages of Function Coverage Testing**

* Functions are easier to identify in a program and hence it is easier to write test cases to provide function coverage
* Since functions are at a higher level of abstraction, its easier to get 100% function coverage than any of the above methods.
* Functions have a direct mapping to requirement and hence function coverage gives a direct indication of requirement coverage also
* It is easier for us to prioritize and test the functions based on their importance

**On a flow graph:**

* Arrows called edges represent flow of control
* Circles called nodes represent one or more actions
* Areas bounded by edges and nodes called regions
* A predicate node is a node containing a condition
* Any procedural design can be translated into a flow graph.
* Note that compound Boolean expressions at tests generate at least two predicate node and additional arcs.



**Basis Path Testing - Deriving Cyclomatic Complexity**

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

**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

**Steps to arrive at Cyclomatic Complexity**

* Draw a corresponding flow graph
* Determine Cyclomatic Complexity
* Determine independent paths
* Prepare tests cases

Cyclomatic Complexity - Example PROCEDURE SORT

1. Do while records remain

read record

2. If record field 1=0

3. Then process record;

store in buffer,

increment counter,

4. Elsif record field 2=0

5. Then reset record;

6. Else process record;

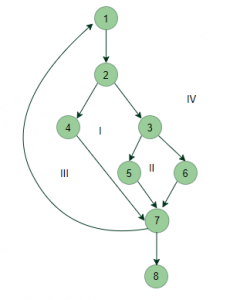
store in file,

7a endif

endif

7b.enddo

8. end



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.

**Advantages:**

* White box testing is very thorough as the entire code and structures are tested.
* It results in the optimization of code removing error and helps in removing extra lines of code.
* It can start at an earlier stage as it doesn’t require any interface as in case of black box testing.
* Easy to automate.

**Disadvantages:**

* Main disadvantage is that it is very expensive.
* Redesign of code and rewriting code needs test cases to be written again.
* Testers are required to have in-depth knowledge of the code and programming language as opposed to black box testing.
* Missing functionalities cannot be detected as the code that exists is tested.
* Very complex and at times not realistic.

**What is Basis Path Testing?**

Basis path testing, a structured testing or white box testing technique used for designing test cases intended to examine all possible paths of execution at least once. Creating and executing tests for all possible paths results in 100% statement coverage and 100% branch coverage.

Example:

Function fn\_delete\_element (int value, int array\_size, int array[])

{

1 int i;

location = array\_size + 1;

2 for i = 1 to array\_size

3 if ( array[i] == value )

4 location = i;

end if;

end for;

5 for i = location to array\_size

6 array[i] = array[i+1];

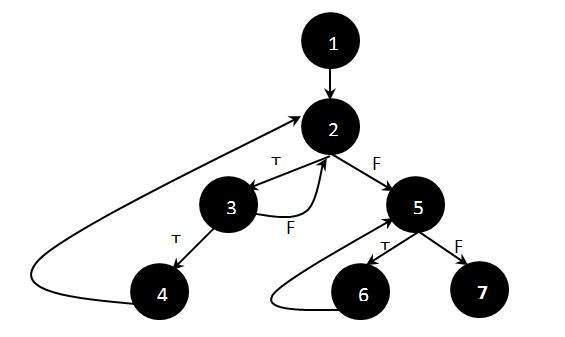
end for;

7 array\_size --;

}

Steps to Calculate the independent paths

Step 1 : Draw the Flow Graph of the Function/Program under consideration as shown below:



Step 2 : Determine the independent paths.

Path 1: 1 - 2 - 5 - 7

Path 2: 1 - 2 - 5 - 6 - 7

Path 3: 1 - 2 - 3 - 2 - 5 - 6 - 7

Path 4: 1 - 2 - 3 - 4 - 2 - 5 - 6 – 7

**Basis Path Testing - Graph Matrices**

Can automate derivation of flow graph and determination of a set of basis paths.

Software tools to do this can use a graph matrix.

**Graph matrix:**

* is square with #sides equal to #nodes
* Rows and columns correspond to the nodes
* Entries correspond to the edges

Can associate a number with each edge entry.

Use a value of 1 to calculate the Cyclomatic complexity

* For each row, sum column values and subtract 1
* Sum these totals and add 1

Interesting link weights are

* Probability that a link (edge) will be executed
* Processing time for traversal of a link
* Memory required during traversal of a link
* Resources required during traversal of a link

**Structured Testing - Data Flow Testing**

* Data flow testing looks at the lifecycle of a particular piece of data (ie a variable) in an application
* Variables that contain data are created, used and killed (destroyed)
* Concerned with the flow of data in the program
* By looking at patterns of data usage, risky areas of code can be found and more test cases can be applied.
* Dataflow testing uses control flow graphs to explore the unreasonable things that can happen to data.

Data can be used in 2 ways – Defined and Used

**Data can be defined.**

**Example of Defined Data (Def)**

int x;

x= a+b;

scanf(&x, &y);

x[i-1] = a+b;

**Data can be used in a variable for performing some computations**

**Example of Used Data (Use)**

a= x+2; (data in x is being used for calculations)

printf(“value of x =“, x);

if (x<10)

* Select paths thought the program’s control flow and test the status of data in each of these paths.
* Pick enough paths to ensure that every data object has been initialized prior to used or all defined objects have been used for something.
* All the def criteria (for definitions of all variables) must be exercised
* All the use criteria of all variable definitions must be covered