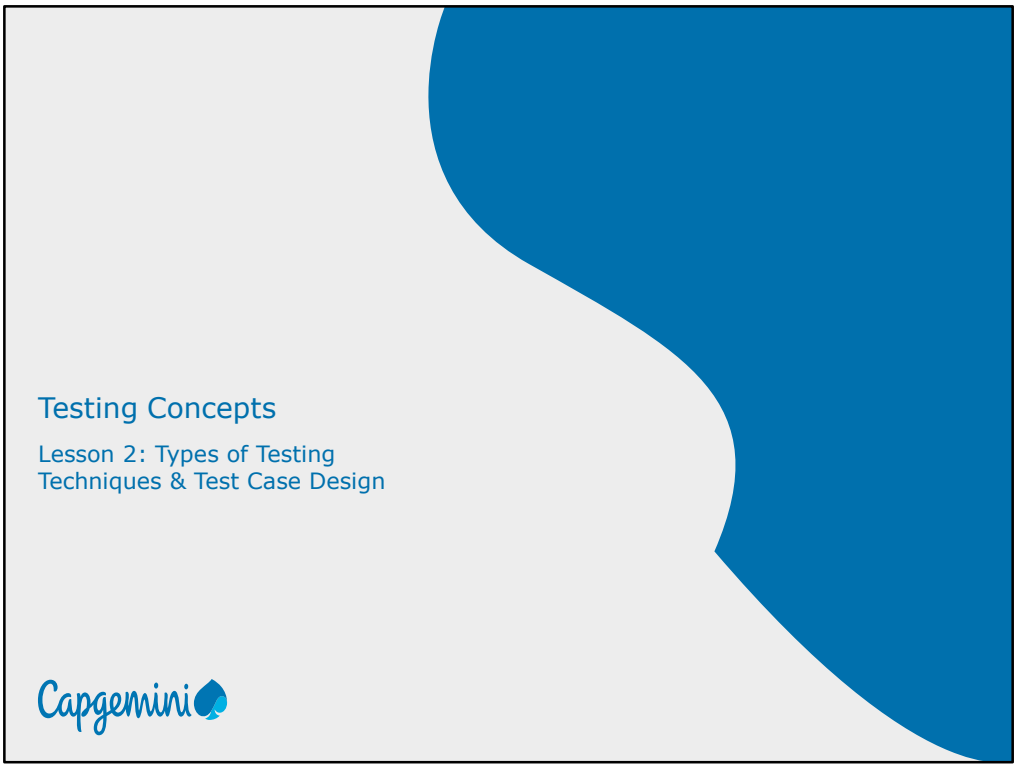

Testing Concepts
Lesson 2: Types of Testing
Techniques & Test Case Design
Capgemini

Lesson Objectives

To understand the following topics:

- Verification and Validation
- Types of Testing Techniques
- Static & Dynamic Testing Techniques
- Introduction to Static Testing Techniques
- Static Testing Techniques – Defects Detected & Benefits
- Review Process Success Criteria
- Introduction to Dynamic Testing
- Types of Dynamic Testing Techniques
- White Box Test Techniques
- Black Box Testing
- Static vs. Dynamic Testing
- A good Test Case
- Test Case Lifecycle
- Test Case Design Techniques

Lesson Objectives

To understand the following topics:

- What is test data?
- Properties of Good Test Data
- Test Data team
- Test data lifecycle
- Requirement and Planning
- Request Process
- Test Data Creation Techniques
- Test Data From Production Data
- Test Data Life Cycle - Maintenance
- Test Data in STLC - Staggered with test case Design
- Test data in STLC -Standalone phase between Test Case Design and Test Case Execution

Lesson Objectives

To understand the following topics:

- What is Positive Testing?
- Advantages/Limitations of positive testing
- What is negative testing?
- Advantages/Limitations of negative testing
- Positive & Negative test scenarios
- What is Basic test?
- Example on Basic test
- What is Alternate test?
- Example on Alternate test
- Importance of writing positive, negative, basic, alternate test while designing test cases
- Best practices for test case maintenance

Verification and Validation

Verification

- Verification refers to a set of activities which ensures that software correctly implements a specific function.
- Purpose of verification is to check: Are we building the product right?
- Example: code and document reviews, inspections, walkthroughs.
- It is a Quality improvement process.
- It is involve with the reviewing and evaluating the process.
- It is conducted by QA team.
- Verification is Correctness.

V&V encompasses many of the activities that are encompassed by S/w quality assurance that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, installation testing.
Example of Verification : code and document inspections, walkthroughs, and other techniques. unit testing , integration testing , system testing

If we are in a shopping centre and buy a thing with a code number 2342 and when we go to till and they check the number of that item and find it wrong then system will check all product number of the relevant number but don't find any number of this kind then we can say that the verify thing is wrong.
Verification is a process, which performs testing to ensure implemented functions meeting to designed functions.

Verification and Validation (cont.)

Validation

- Purpose of Validation is to check : Are we building the right product?
- Validation refers to a different set of activities which ensures that the software that has been built is traceable to customer requirements.
- After each validation test has been conducted, one of two possible conditions exist:
  - 1. The function or performance characteristics conform to specification and are accepted, or
  - 2. Deviation from specification and a deficiency list is created.

Example : a series of black box tests that demonstrate conformity with requirements.

- It ensures the functionality.
- It is conducted by development team with the help from QC team.
- Validation is Truth.
- Validation is the following process of verification.

Validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements.

Types of Testing Techniques

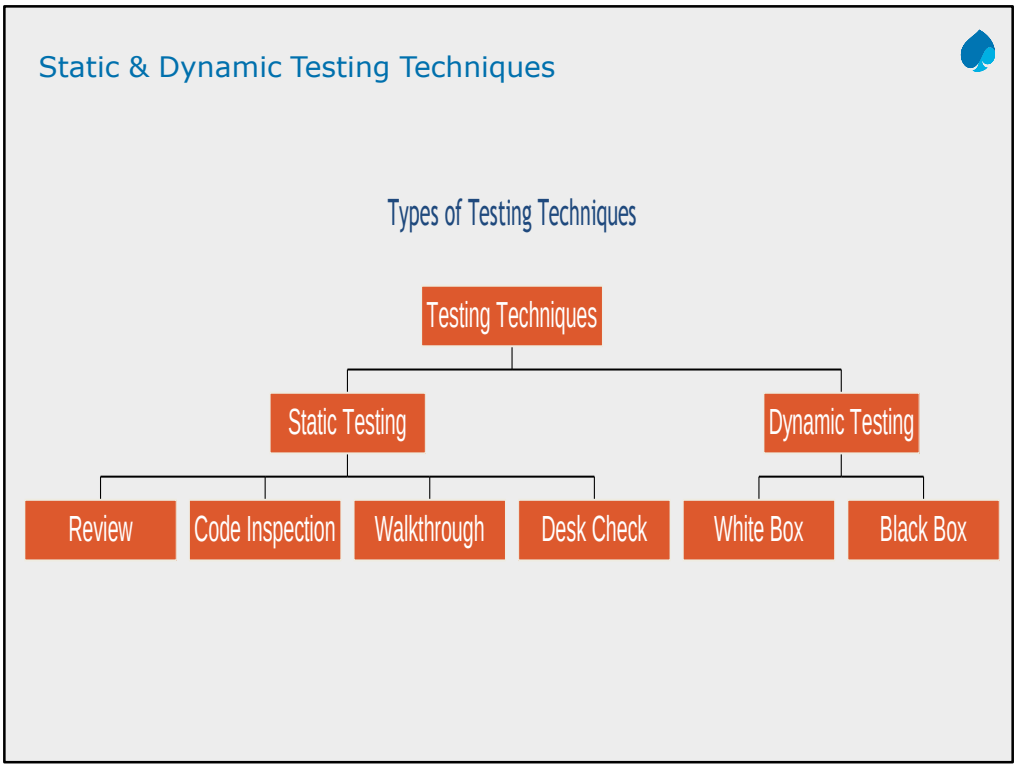
Static Testing

- It is a ***verification*** process
- Testing a software without execution on a computer. Involves just examination/review and evaluation
- It is done to test that software confirms to its SRS i.e. user specified requirements
- It is done for ***preventing*** the defects

Dynamic Testing

- It is a ***validation*** process
- Testing software through executing it
- It is done to test that software does what the user really requires
- It is done for ***detecting*** the defects

Static & Dynamic Testing Techniques



Introduction to Static Testing Techniques

Static Testing is a process of reviewing the work product and reviewing is done using a checklist

Static Testing helps weed out many errors/bugs at an early stage

Static Testing lays strict emphasis on conforming to specifications

Static Testing can discover dead codes, infinite loops, uninitialized and unused variables, standard violations and is effective in finding 30-70% of errors

Static Testing Methods

- Self Review
- Code Inspection
- Walk Through
- Desk Checking (Peer Review)

Introduction to Static Testing Techniques

How can we evaluate or analyze a requirements document, a design document, a test plan, or a user manual or examine a source code?
By reviewing those.
These static techniques rely on manual examinations (Reviews) and automated Analysis (Static Analysis) without execution
Any software product can be reviewed
Purpose of reviews
Finding defects
informational, communicational and educational

Self Review

Self review is done by the person who is responsible for a particular program code

It is more of reviewing the code in informal way

It is more like who writes the code, understands it better

Self review is to be done by the programmer when he builds a new code

There are review checklists that helps programmer to verify with the common errors regarding the program code

ERROR CHECKLIST FOR INSPECTION

1. Data Reference Errors

For each array reference, is each subscript value within defined bounds?

Dangling reference problem: arises when the lifetime of a pointer is greater than the lifetime of a referenced storage.

If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?

2. Data Declaration errors

Is each variable has been assigned correct length, type, and storage class?

Are there any variables with similar names (not an error but may have been confused with in the program)?

3. Computation errors

Are there any computations using same data types but different lengths?

Is an underflow or overflow occurs during computation?

Division by zero and square root of a negative number errors.

Are the order of evaluation and precedence of operators correct?

4. Comparison errors:

Are there any mixed mode computations or comparisons between variables of different lengths?

Is the comparison operator correct? Does each Boolean expression state what it is supposed to do? Are the operands of a Boolean operator Boolean?

Code Review Checklist

Data Reference Errors
- Is a variable referenced whose value is unset or uninitialized?

Data Declaration Errors
- Have all variables been explicitly declared?
- Are variables properly initialized in declaration sections?

Computation errors
- Are there any computations using variables having inconsistent data types?
- Is there any mixed mode computations?

Comparison errors
- Are there any comparisons between variables having inconsistent data types?

Control Flow errors
- Will every loop eventually terminate?
- Is it possible that, because of condition upon entry, a loop will never execute?

Interface errors
- Does the number of parameters received by these module equals the number of arguments sent by calling modules?
- Also is the order correct?

Input/output errors
- All I/O conditions handled correctly?

ERROR CHECKLIST FOR INSPECTION

1. Data Reference Errors

For each array reference, is each subscript value within defined bounds?
Dangling reference problem: arises when the lifetime of a pointer is greater than the lifetime of a referenced storage.
If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?

2. Data Declaration errors

Is each variable has been assigned correct length, type, and storage class?
Are there any variables with similar names (not an error but may have been confused with in the program)?

3. Computation errors

Are there any computations using same data types but different lengths?
Is an underflow or overflow occurs during computation?
Division by zero and square root of a negative number errors.
Are the order of evaluation and precedence of operators correct?

4. Comparison errors:

Are there any mixed mode computations or comparisons between variables of different lengths?
Is the comparison operator correct? Does each Boolean expression state what it is supposed to do? Are the operands of a Boolean operator Boolean?

Code Inspection

Code inspection is a set of procedures and error detection techniques for group code reading.

Involves reading or visual inspection of a program by a team of people,

hence it is a group activity.

The objective is to find errors but not solutions to the errors

An inspection team usually consists of:

- A moderator
- A programmer
- The program designer
- A test specialist

Moderator duties includes:

Distribution materials,
Scheduling the Inspection Session
Leading the session
Recording all errors found
Ensures that the errors found are subsequently corrected.

Code Inspection

Before the Inspection

- The moderator distributes the program's listing and design specification to the group well in advance of the inspection session

During the inspection

- The programmer narrates the logic of the program, statement by statement
- During the discourse, questions are raised and pursued to determine if errors exist
- The program is analyzed with respect to a check list of historically common programming errors

Code Inspection Helps in

- Detect Defects
- Conformance to standards/spec
- Requirements Transformation into product

Code inspection focuses on discovering errors and not correcting them. The Inspection process is the way of identifying error prone sections early, helping to concentrate on the most sensitive sections during testing process.

Code Walkthrough

Code Walkthrough is a set of procedures and error detection techniques for group reading.

Like code inspection it is also an group activity.

In Walkthrough meeting, three to five people are involved. Out of the three, one is moderator, the second one is Secretary who is responsible for recording all the errors and the third person plays a role of Test Engineer.

Solutions are also suggested by team members.

Walkthrough helps in

- Approach to Solution
- Find omission of requirements
- Style / Concepts Issues
- Detect Defects
- Educate Team Members

Walkthrough meeting.

This meeting will includes following members:
A highly experienced programmer
A programming language expert
A new programmer
The person who maintains the program
Some person from a different project
Some one from the same team as a programmer.

Walk through procedure
The designer simulates the program.
She/he shows, step by step what the program will do with the test data supplied by the reviewers.
The simulation shows how different pieces of the system interact and can expose awkwardness, redundancy and many missed details.
Different from inspection that it needs the participants to be ready with test cases.

Desk Checking (Peer Review)

- Human error detection technique
- Viewed as a one person inspection or walkthrough
- A person reads a program and checks it with respect to an error list and/or walks test data through it
- Less effective technique
- Best performed by the person other than the author of the program

It is the dry run of the program. It is completely informal process. Desk checking is best performed by a person other than the author of the program.
E.g. Two programmers might swap the program rather than desk checking their own programs.
It is less effective than inspection and walkthrough of the code.

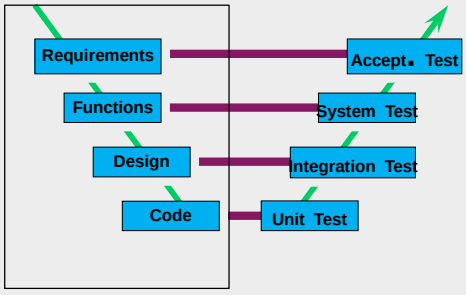
Static Testing Techniques – Defects Detected & Benefits

Types of defects found during static reviews:

- Deviations from standard
- Requirement defects
- Design defects
- Insufficient maintainability
- Incorrect interface specification

Benefits of Reviews:

- Early feedback on quality
- Development productivity improvement
- Reduced development timescales
- Reduced testing time and cost
- Lifetime cost reductions
- Reduced fault levels
- Increased awareness of quality issues



It is the dry run of the program. It is completely informal process. Desk checking is best performed by a person other than the author of the program.
E.g. Two programmers might swap the program rather than desk checking their own programs.
It is less effective than inspection and walkthrough of the code.

Review Process Success Criteria

Success factors for reviews:
Review should be with clear objective
Defects found are welcomed and expressed objectively
Management supports a good review process
The emphasize is on learning and process improvement
The right people for review are involved
Appropriate use of review techniques
Reviews are conducted in a fair & trustworthy atmosphere
Explicitly planning and tracking review activities
Train the participants in the review techniques – particularly the formal ones such as inspection

It is the dry run of the program. It is completely informal process. Desk checking is best performed by a person other than the author of the program.
E.g. Two programmers might swap the program rather than desk checking their own programs.
It is less effective than inspection and walkthrough of the code.

Introduction to Dynamic Testing

Dynamic Testing involves working with the software, giving input values and validating the output with the expected outcome

Dynamic Testing is performed by executing the code

It checks for functional behavior of software system , memory/CPU usage and overall performance of the system

Dynamic Testing focuses on whether the software product works in conformance with the business requirements

Dynamic testing is performed at all levels of testing and it can be either black or white box testing

It is the dry run of the program. It is completely informal process. Desk checking is best performed by a person other than the author of the program.
E.g. Two programmers might swap the program rather than desk checking their own programs.
It is less effective than inspection and walkthrough of the code.

Types of Dynamic Testing Techniques

White Box Test Techniques

- Code Coverage
  - Statement Coverage
  - Decision Coverage
  - Condition Coverage
  - Loop Testing
- Code complexity
  - Cyclomatic Complexity
- Memory Leakage

Black Box Test Techniques

- Equivalence Partitioning
- Boundary Value Analysis
- Use Case / UML
- Error Guessing
- Cause-Effect Graphing
- State Transition Testing

White Box Test Techniques

White box is logic driven testing and permits Test Engineer to examine the internal structure of the program

Examine paths in the implementation

Make sure that each statement, decision branch, or path is tested with at least one test case

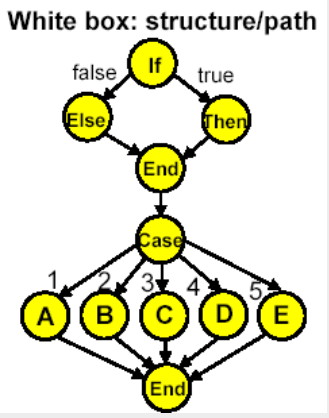
Desirable to use tools to analyze and track Coverage

White box testing is also known as structural, glass-box and clear-box

White Box Test Techniques

White Box Test Techniques

- Code Coverage
  - Statement Coverage
  - Decision Coverage
  - Condition Coverage
  - Loop Testing
- Code complexity
- Memory Leakage



Using white box testing methods, you can derive test cases that

guarantee that all independent paths within a module have been excised 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 ensure their validity.

Code Coverage

Measure the degree to which the test cases exercise or cover the logic (source code) of the program

Types

- Statement Coverage
- Decision Coverage
- Conditional Coverage
- Loop Testing

Code coverage is also known as logic coverage. The goal is to execute every statement of the code at least once. Test engineers can derive test cases using

White box testing methods, that

All independent paths within a module are traversed at least once
Exercise all logical decisions on their true and false sides
Execute all loops at their boundaries and within operational bounds
Exercise internal data structures to ensure their validity.

Statement Coverage

Test cases must be such that all statements in the program is traversed at least once

Consider the following snippet of code

void procedure(int a, int b, int **x)**

```
{   If (a>1) && (b==0)
            { x=x/a;        }
    If (a==2 || x>1)
            {  x=x+1; }
}
```

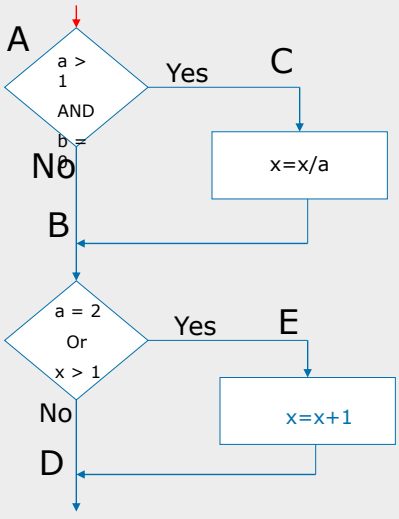
The goal is to execute every statement in the program at least once. Every statement must be executed.

Statement Coverage

Test Case: a=2,b=0, x=3.

Every statement will be executed once.

But only path ACE will be covered and path ABD,ACD,ABE will not be covered.



Every statement can be executed by writing a single test case. This case covers only ACE path.
This criteria is weak one. Since it is not considering other paths to traverse. So the path ABD, ACD, ABE would go undetected.

Statement Coverage

In the above code one test case is sufficient to execute each of the two if statements at least once:

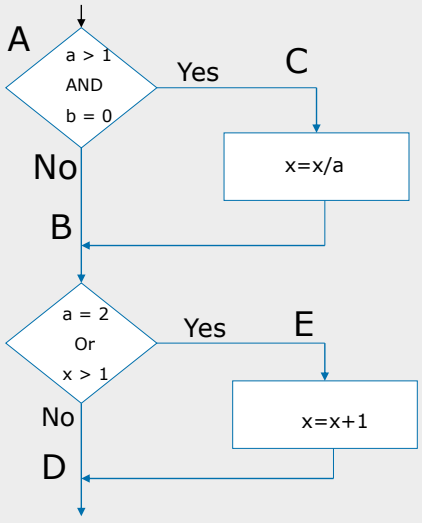
Test Case : a=2 , b=0 , x=3

(Decision1 is True, Decision2 is True)

However this test case does not help in detecting many of the many of the bugs which may go unnoticed as the false outcomes of the conditions a>1 & b=0 , a=2 or x>1 are not tested

Decision Coverage

Test Case 1: a=2, b=0, x>1
(Decision1 is True, Decision2 is True) (Path ACE)
Test Case 2: a<=1 , b!=0, x<=1
(Decision1 is False, Decision2 is False) (Path ABD)



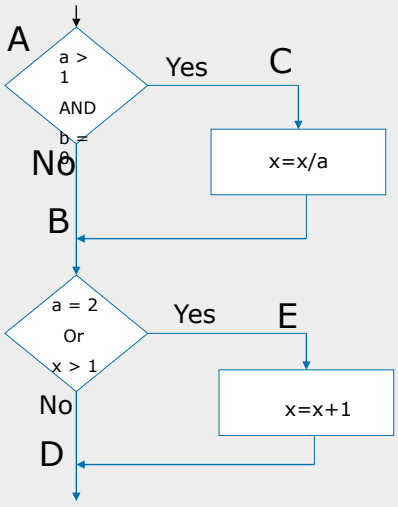
Decision coverage can cover two test cases covering paths ACE and ABD. Even if the above test cases satisfy decision coverage it still does not cover the path ACD and path ABE. Hence decision coverage though stronger criteria than statement it is still weak. There is only 50 percent chance that we would explore the path.

Condition Coverage

Test cases are written such that each condition in a decision takes on all possible outcomes at least once.

Test Case1 : a=2, b=0, x=3
(Condition1 is True,Condn2 is True)
(Path ACE)

Test Case2: a=3, b=0, x=0
(Condn1 is True,Condn2 is False,Condn3 is False)
(Path ACD)



Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression. Relational operator is one of the following <, <=, =, not =, > , =>.

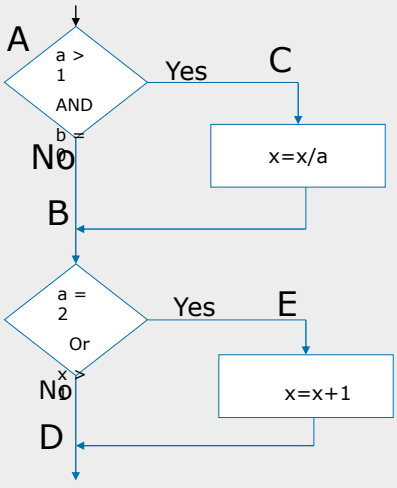
A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.

Condition coverage focuses on testing each condition in a program. The purpose of the condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

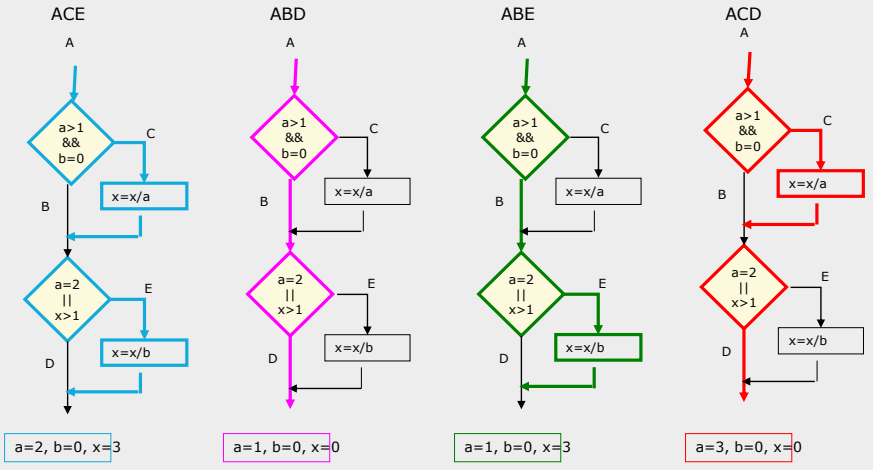
Condition Coverage

Test Case3 : a=1, b=0, x=3
(Condition1 is False,Condition2 is True)
(Path ABE)

Test Case4: a=1, b=1, x=1
(Condition1 is False,Condition2 is False)
(Path ABD)



Condition Coverage



What does “coverage” mean?

- NOT all possible combinations of data values or paths can be tested
- Coverage is a way of defining how many of the paths were actually exercised by the tests
- Coverage goals can vary by risk, trust, and level of test

In the above diagrams, each condition in decision takes all possible outcomes at least once.

Loop Testing

Loops testing is a white box testing technique that focuses exclusively on validity of Loop construct

Types of loops

- Simple Loop
- Nested Loop
- Concatenated Loop
- Spaghetti Loop

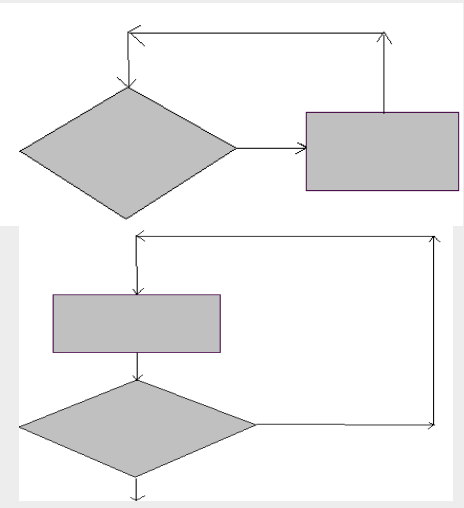
Loops are cornerstones for may algorithms. It is very important to test every loop carefully.

Loop Testing

Simple Loop Testing Procedure:

- skip the entire loop
- only one pass through the loop
- make 2 passes through loop
- m passes through loop where m<n
- n-1, n, n+1 passes through the loop

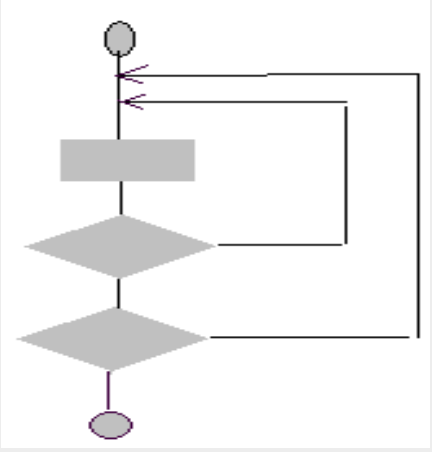
Where n is the maximum number of allowable passes through the loop



Loop Testing

Nested Loop Testing Procedure:

- start at the innermost loop
- conduct simple loop test for the innermost loop
- work outward, conducting tests for the
- next loop but keeping all other loops at minimum
- continue until all the outer loops are tested

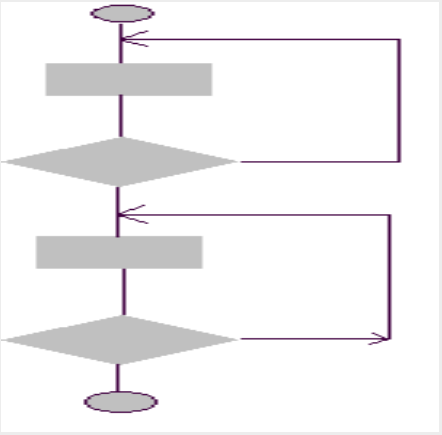


Number of test cases increases as the loop gets extended.

Loop Testing

Concatenated Loop Testing Procedure:

- If each loop is independent of the other, test them as simple loops, else test them as nested loops

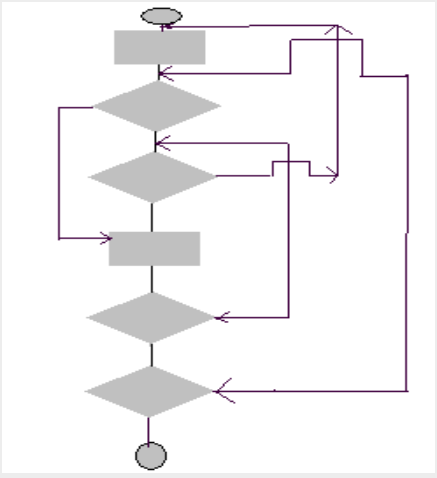


If the loops are independent of others, concatenated loops can be tested using simple loop testing approach. However, if the loops are concatenated and are not independent, the loop counter for loop 1 is used as the initial value for loop 2.

Loop Testing

Spaghetti loops Testing Procedure:

- Redesign using structured constructs



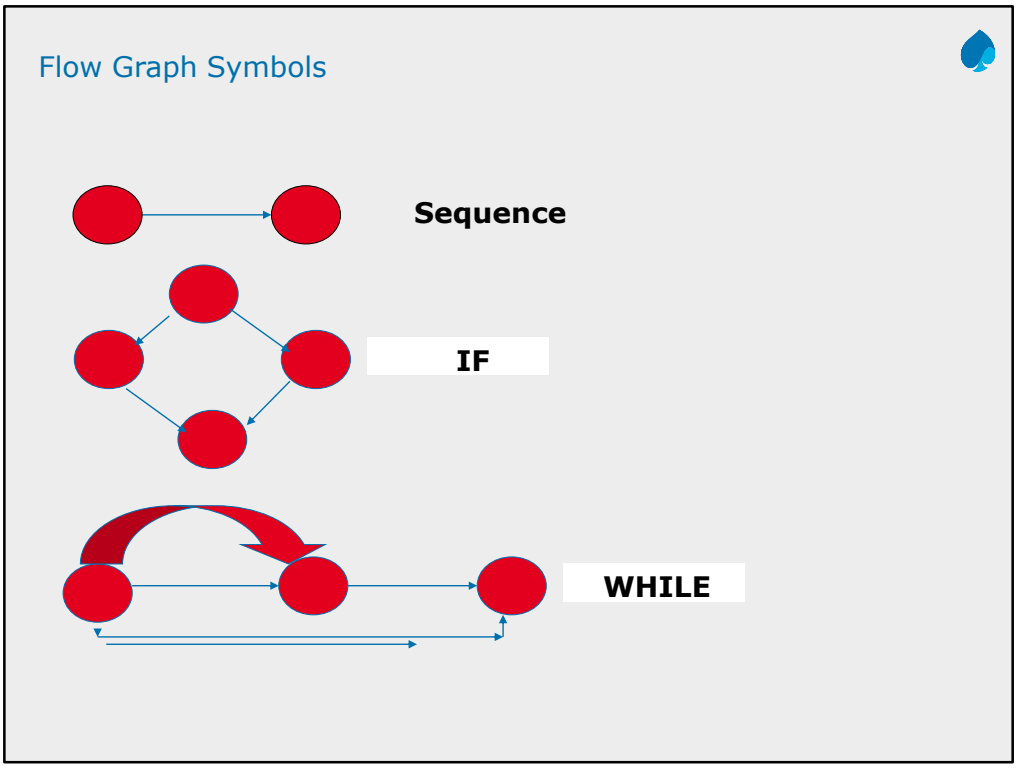
These are the unstructured loops. These loops should be redesigned to structured constructs.

Flow Graph

- Main tool for test case identification
- Shows the relationship between program segments , which is the sequence of statements having the property that if the first member of the sequence is executed then all other statements in that sequence will also be executed

Flow Graph Symbols

- Nodes represent one program segment
- Areas bounded by edges and nodes are called regions
- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition
- Each node containing a condition is called a predicate node



Sequence Flow – indicates the statements to be executed in a defined sequence
If – Sequence of statements will be executed depending on the condition match
While - Sequence of statements will be executed depending on the condition match, another way of achieving If flow.

Flow graph is the notation for the representation of the control flow. It depicts logical control flow with the help of notations.
Each circle is called a flow graph node, represents one or more procedural statements. The arrows on the flow graph , called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements.
Areas bounded by edges and nodes are called regions. When counting regions, we include the area out-side the graph as a region.
When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
Each node containing a condition called a predicate node. It is characterized by two or more edges emanating from it.

Cyclomatic Complexity

- Cyclomatic Complexity (Code Complexity) is a software metric that provides a quantitative measure of logical complexity of a program
- When Used in the context of the basis path testing method, value for cyclomatic complexity defines number of independent paths in basis set of a program
- Also provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once
- Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity

Introduced by Thomas McCabe in 1976:

Count the regions of the flow graph (including the exterior)
Or compute by e-n+2 This is called Cyclomatic Complexity
The number of paths to test , all decision options are tested

How many paths (McCabe's technique for units)?

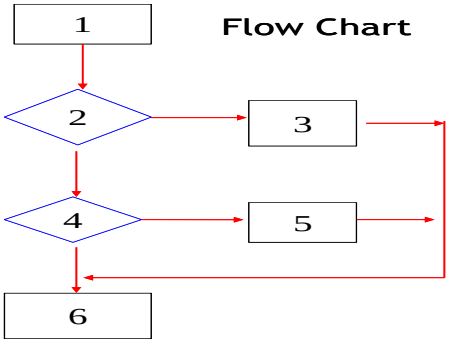
Cyclomatic complexity defines the number of independent paths. This provides minimum number of tests to be conducted to ensure all the statements have been executed at least once.
An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

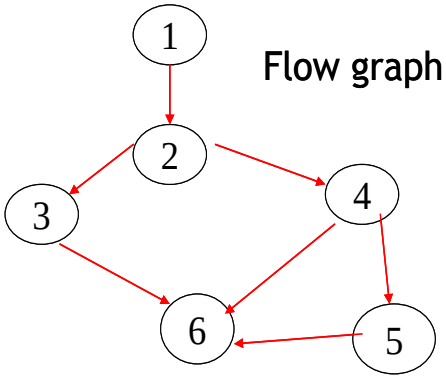
Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. Use it for test planning as well as test case design.
Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
When used in context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us the upper bound of the number of tests that must be conducted to ensure that all statements have been executed at least once.

Calculating Cyclomatic Complexity

- The cyclomatic complexity of a software module is calculated from a flow graph of the module , when used in context of the basis path testing method
- Cyclomatic Complexity V(G) is calculated one of the three ways:
  - V(G) = E - N + 2 , where E is the number of edges and N = the number of nodes of the graph
  - V(G) = P+1, where P is the number of predicate nodes
  - V(G) = R , where number of region in the graph

Here, a flow chart is used to depict program control structure. Flow chart is mapped into corresponding flow graph. Each circle is called as flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node.



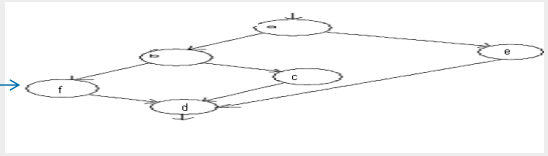


A flow chart depicts program control structure.
Flow chart is mapped into flow graph.
A sequence of process boxes and a decision diamond are map into a single node.

Each circle is called as graph node.
Arrows are called as edges.
The arrows on the flow graph is called edges, represents flow of control. An edge must be terminated at a node.

Calculating Cyclomatic Complexity : Example

In the given figure a and b are predicate nodes



1. Cyclomatic Complexity, V(G) for a flow Graph G is V(G) = E – N + 2

E = Number of Edges in the graph (7 in the above figure)

N = number of flow graph Nodes (6)

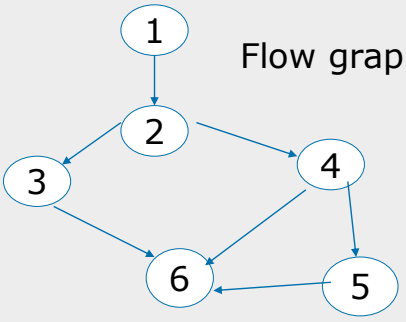
R = number of Regions (3)

Hence V(G) = 7-6+2 = 3

2. V(G) can also be calculated as V(G) = P+1, where P is the number of predicate nodes. Here V(G) = 2+1 = 3
3. Also V(G) can be calculated as V(G) = R hence V(G) = 3

The value of V(G) provides the number of linearly independent paths through the program.
Here, in the above graph, value of V(G) is 3. That is, minimum 3 test cases must be designed to guarantee coverage of all program statements.

Calculating Cyclomatic Complexity : Example



Flow graph

Edges(E) = 7 Nodes(N) = 6

Regions (R) = 3 , Predicate nodes (P) = 2

CC = E - N + 2 = 7- 6 + 2 = 3

CC = P + 1 = 2 + 1 = 3

CC = R = 3

Cyclomatic complexity gives minimum number of test cases to be performed to cover all the program statements.

In the above graph, the cyclomatic complexity is 3. That means, there are 3 independent paths in this program. Independent path is any path through the program that has set of statements or a condition.
Path 1: 1-2-3-6
Path 2: 1-2-4-5-6
Path 3: 1-2-4-6
In order to have complete coverage of this code, there are minimum 3 test cases are required which traversed thorough the above paths.

Memory Leak

Memory leak is present whenever a program loses track of memory.
Memory leaks are most common types of defect and difficult to detect
Performance degradation or a deadlock condition occurs
Memory leak detection tools help to identify

- memory allocated but not deallocated
- uninitialized memory locations

Memory Leak

Find the error in the following snippet of code

```
void read_file(char*);
void test(bool flag)
{
      char* buf = new char[100];
       if (flag) {
                read_file(buf);
                delete [] buf;
                }
 }
```

This problem occurs if a program fails to free objects that are no longer in use. The code leaks 100 bytes of memory every time the function test is called with the argument flag as false.
If a program continues to leak memory, its performance degrades. Its runtime memory footprint continues to increase and it spends more and more time in swapping and can eventually run out of memory.
You can use the garbage collection library to fix these errors.

Same memory pointer is deleted more than once:

```
void test3()
{
 int* p1 = new int;
 int* p2 = p1;
 delete p1;
 delete p2;  // deleting again!
}
```

Memory Fragmentation and Overwrites

Memory Fragmentation

- caused by frequent allocation and deallocation of memory
- can degrade an application's performance
- occurs when a large chunk of memory is divided into much smaller, scattered pieces
- May not be never allocated again

Memory Overwrites

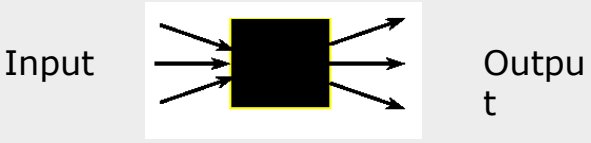
- too little memory is allocated for an object
- can include memory corruption and intermittent failures.
- program may work correctly some times and fail at other times

Memory Fragmentation
This can be caused by frequent allocation and deallocation of memory and can degrade an application's performance. Memory fragmentation occurs when a large chunk of memory is divided into much smaller, scattered pieces. These smaller discontiguous chunks of memory may not be of much use to the program and may never be allocated again. This can result in the program consuming more memory than it actually allocates.
Memory Overwrites
This problem occurs when too little memory is allocated for an object. Consequences can include memory corruption and intermittent failures. The program may work correctly some times and fail erratically at other times.

Memory Overwrites: Once a block of memory has been allocated, it is important that the program does not attempt to write any data past the end of the block or write any data just before the beginning of the block. Even writing a single byte just beyond the end of an allocation or just before the beginning of an allocation can cause disaster. It is a possible candidate for turning on overflow buffers.

Black Box Testing

- Black box is data-driven, or input/output-driven testing
- The Test Engineer is completely unconcerned about the internal behavior and structure of program
- Black box testing is also known as behavioral, functional, opaque-box and closed-box



Black Box At Different Levels – Unit, Subsystem and System.
A black box is just a bigger box with more input, functionality, and output.

Black Box Testing

Trainer Notes

Black Box Testing

- Tests are designed to answer the following questions:
- How is functional validity tested ?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- What effect will specific combinations of data have on system operations?

Black Box testing also called behavioral testing, focuses on the functional requirements of the software. Black box testing is not an alternative to white box techniques. Rather it is complementary approach that is likely to uncover a different class of errors than white box methods.

Black box testing 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 errors.

Black Box Test Techniques

There are various techniques to perform Black box testing ;

- Equivalence Partitioning
- Boundary Value Analysis
- Error Guessing
- Cause Effect Graphing
- State transition testing

Programmers are logical thinkers, so they catch many of the “logical’ defects. Real users are NOT necessarily logical.
Real environmental circumstances are often illogical.

Equivalence Partitioning

This method divides the input domain of a program into categories of data for deriving test cases

Identify equivalence classes - the input ranges which are treated the same by the software

- Valid classes: legal input ranges
- Invalid classes: illegal or out of range input values

The aim is to group and minimize the number of test cases required to cover these input conditions

Assumption:

If one value in a group works, all will work

One from each partition is better than all from one

Thus it consists of two steps:

- Identify the Equivalence class
- Write test cases for each class

For those familiar with elementary statistical techniques, EP is very much similar to class intervals and tally marks analysis.

An ideal test case single handedly uncovers a class of errors (e.g. incorrect processing of all character data) that might require many cases to be executed before general error is observed. EP strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

An equivalence class represents a set of valid or invalid states for input condition. An input condition is either a specific numeric value , a range of values , a set of related values or a Boolean condition.

Equivalence Partitioning

Examples of types of equivalence classes

If an input condition specifies a continuous range of values, there is one valid class and two invalid classes

Example: The input variable is a mortgage applicant's income. The valid range is $1000/mo. to $75,000/mo

- Valid class: {1000 > = income < = 75,000}
- Invalid classes: {income < 1000}, {income > 75,000}

If an input condition specifies a set of values, there is reason to believe that each is handled differently in the program.

e.g., Type of Vehicle must be Bus, Truck, Taxi). A valid equivalence class would be any one of the values and invalid class would be say Trailer or Van.

Equivalence Partitioning

If an input condition specifies that a variable, say count, can take range of values(1 - 999)

Identify - one valid equivalence class (1 < count < 999)
- two invalid equivalence classes (count < 1) & (count >999)

Equivalence classes may be defined according to the following guidelines.

If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
If an input condition requires a specific value, one valid and two invalid EC are defined.
If an input condition specifies a member of a set, one valid and one invalid EC are defined.
In an input condition is Boolean, one valid and one invalid class are defined.

Equivalence Partitioning

If a "must be" condition is required, there is one valid equivalence class and one invalid class

Example: The mortgage applicant must be a person

- Valid class: {person}
- Invalid classes:{corporation, ...anything else...}

If we have to test function int Max (int a , int b) the Equivalence Classes for the arguments of the functions will be

Arguments	Valid Values	Invalid Values
A	-32768 <= Value <= 32767	< - 32768 , >32767
B	-32768 <= Value <= 32767	< - 32768 , >32767

Boundary Value Analysis

"Bugs lurk in corners and congregate at boundaries ….." *Boris Beizer*

Boundary Conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes.

Boundary value analysis is a test case design technique that complements Equivalence partitioning.

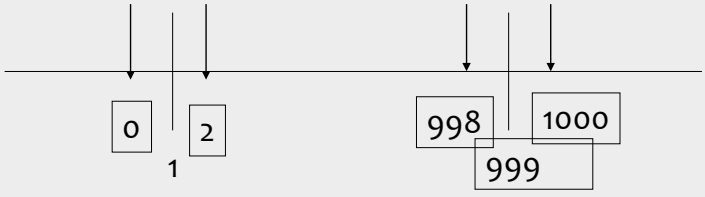
Test cases at the boundary of each input Includes the values at the boundary, just below the boundary and just above the boundary.

BVA is not as simple as it sounds, because boundary conditions may be subtle and difficult to identify.
The method does not test combinations of input conditions.

Boundary Value Analysis

From previous example, we have the valid equivalence class as (1 < count < 999)

Now, according to boundary value analysis, we need to write test cases for count=0, count=1,count=2,count=998,count=999 and count=1000 respectively



General guidelines:
If an input or output condition specifies a range of values, write test cases for the ends of the range.
If an input or output condition specifies a number of values, write test cases for the minimum and maximum number of values.
If the input or output of a procedure is an ordered set, focus attention on the first and last elements of the set.

Guidelines for BVA are similar in many respects to those provided for EP:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b as well as just above and just below.
2. If an input condition specifies a number of values test cases should be developed that exercise the minimum and maximum numbers. Values just above and below min and max are also tested.
3. Applying guidelines 1 and 2 output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to cerate an output report that produces the maximum (and Min) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g.- an array has defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Boundary Value Analysis

Guidelines

- If an input condition specifies a range of values A and B, test cases should be designed with values A and B, just above and just below A and B respectively
- Similarly with a number of values

Boundary Value Analysis

Example :

If we have to test function int Max(int a , int b) the Boundary Values for the arguments of the functions will be

Arguments	Valid Values	Invalid Values
A	-32768, -32767, 32767, 32766	-32769,32768
B	-32768, -32767, 32767, 32766	-32769,32768

Error Guessing

- Based on experience and intuition one may add more test cases to those derived by following other methodologies
- It is an ad hoc approach
- The success of error guessing is very much dependent on the skill of the tester, as good testers know where the defects are most likely to lurk
- This is why error guessing testing technique should be used along with other formal testing techniques
- The basis behind this approach is in general people have the knack of "smelling out" errors
- There are no rules for error guessing
- The tester is encouraged to think of situations in which the software may not be able to cope.

Here mention about Myers Probability study that the probability of errors remaining in the program is proportional to the number of errors that have been found so far, which provides a rich source for productive error guessing.

Error Guessing

Make a list of possible errors or error-prone situations and then develop test cases based on the list

Defects' history are useful

Probability that defects that have been there in the past are the kind that are going to be there in the future

Some examples :

- Empty or null lists/strings
- Zero occurrences
- Blanks or null character in strings
- Negative numbers

Trainer Notes

Error Guessing

Example : Suppose we have to test the login screen of an application

An experienced test engineer may immediately see if the password typed in the password field can be copied to a text field which may cause a breach in the security of the application

Error guessing testing for sorting subroutine situations

- The input list empty
- The input list contains only one entry
- All entries in the list have the same value
- Already sorted input list

Trainer Notes

Cause Effect Graphing

A testing technique that aids in selecting, in a systematic way, a high-yield set of test cases that logically relates causes to effects to produce test cases

It has a beneficial side effect in pointing out incompleteness and ambiguities in specifications

Steps:

- Identify the causes and effects from the specification
- Develop the cause effect diagram
- Create a decision table
- Develop test cases from the decision table

BVA and Equivalence partitioning do not explore combinations of input circumstances.
In Cause effect graphing, causes and effects are identified.
Cause is a distinct input condition. Effect is a distinct output condition.

Cause and Effect - Simple example
E.g. 1
Cause: Got caught in rain
Effect: Cold and cough
E.g. 2
Cause: Hours of Dance practice
Effect: First Prize in the competition

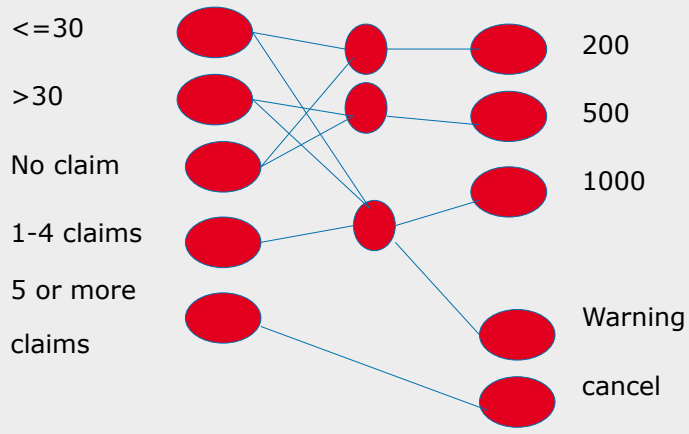
Cause Effect Graphing

Insurance policy renewal example

- An insurance agency has the following norms to provide premium to its policy holders
- If age<=30 and no claim made premium increase will be 200 else 500
- For any age if claims made is 1 to 4 then premium increase will be 1000
- If one or more claims made then warning letter, if 5 or more claims made then cancel policy

It helps to create the cause and effect diagram through the decision table.

Cause Effect Graphing



The above graph covers all the combinations of the scenarios where
If age<=30 and no claim made premium increase will be 200 else 500
For any age if claims made is 1 to 4 then premium increase will be 1000
If one or more claims made then warning letter, if 5 or more claims made then cancel policy

Cause Effect Graphing

Insurance Renewal Decision Table

No. of Claims	Insured Age	Premium Increase	Send Warning	Cancel
0	<=30 >30	200 500	No No	No No
1-4	All ages	1000	Yes	No
5 or more			No	Yes

The cause effect graph is converted into decision table. The entire policy terms get transformed into tabular format in more easier way.

If age<=30 and no claim made premium increase will be 200 else 500. No warning will be sent and no cancellation is done.
For any age if claims made is 1 to 4 then premium increase will be 1000. Warning will be sent but no cancellation is done
If 5 or more claims made then cancel policy

Cause Effect Graphing

Causes	Effects
C1 age<=30 C2 age >=30 C3 No claims C4 1-5 claims C5 >5 claims	E1 Premium increase 200 E2 Premium increase 500 E3 Premium increase by 1000 E4 Warning Letter E5 Cancel Premium

Policy terms are mentioned in the cause and effect manner. All the conditions are termed as causes and its result is reflected as effects in the above table.

State Transition Testing

A testing techniques that aids to validate various states when an program moves from one visible state to another

It is a techniques in which test cases are designed to execute valid and invalid state transition

Menu System Example :

The program starts with an introductory menu. As an option is selected the program changes state and displays a new menu. Eventually it displays some information , data input screen.

Each option in each menu should be tested to validate that each selection made takes us to the state we should reach next.

For Example:
A registration form as two buttons, viz. OK and CANCEL. After filling up the entire application, OK button changes to SAVE caption to save the filled data.
So the single button is acting in two different ways depending on its state transition, which has to be tested.

State Transition Testing

Washing machine has different modes like soak, wash, rinse & dry

Machine in these different states, exhibit different features

- Soak mode - clothes absorb soap water
- Wash mode - clothes get washed with soap water
- Rinse mode - It removes soup water from clothes
- Dry mode - water gets removed from clothes

It is useful to create a state transition diagram to spot relationship between states and trace transition between states

Example :
Drawn above is a classic example of State Transition testing. It depicts about the Stack implementation.

Initially Stack is in Empty state.
As soon as, some elements get added to it with push() method, its state changes to Loaded.
When the element reaches to the Stack's maximum capacity, its state changes from Loaded to FULL.
Similarly, while removing or accessing from the Stack with pop() method, it gets the element into Loaded state. It extracts the topmost element.

Static vs. Dynamic Testing

Static Testing	Dynamic Testing
It is the process of confirming whether the software meets its requirement specification	It is the process of confirming whether the software meets user requirements.
Examples : Inspections, walkthroughs and reviews	Examples : structural testing, black-box testing, integration testing, acceptance testing
It is the process of inspecting without executing on computer	It is the process of testing by executing on computer
It is conducted to prevent defects	It is conducted to correct the defects
It can be done before compilation	It takes place only after compilation and linking

Introduction – Test Case Construction & Test Data Preparation

Test cases construction and test data preparation are the first stages of testing

Test cases are prepared based on test ideas

"A test idea is a brief statement of something that should be tested. "

- For example, if you're testing a square root function, one idea for a test would be 'test a number less than zero'

" The idea of preparing a test case is to check if the code handles an error case."

Designing good test cases is a complex art. The complexity comes from three sources:

Test cases help us discover information. Different types of tests are more effective for different classes of information.

Test cases can be "good" in a variety of ways. No test case will be good in all of them.

People tend to create test cases according to certain testing styles, such as domain testing or risk-based testing. Good domain tests are different from good risk-based tests.

Test Case

Test Case is a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

In other words, Test Case is a planned sequence of actions (with the objective of finding errors)

Test cases may be designed based on -

- Values – Valid/Invalid/Boundary/Negative
- Test conditions

Test case will be complex if there is more than one expected result.

A test case is a question that we ask the program. The point of running the test is to gain information, for example whether the program will pass or fail the test.
Characteristics of a Good Test:
They are: likely to catch bugs
not redundant
not too simple or too complex.
Test case is a triplet [I, S, O] where
I is input data
S is state of system at which data will be input
O is the expected output

Test Case Terminologies

Pre Condition

- Environmental and state which must be fulfilled before the component/unit can be executed with a particular input value.

Test Analysis

- is a process for deriving test information by viewing the Test Basis
- For testing, test basis is used to derive what could be tested

Test basis includes whatever the test are based on such as System Requirement

- A Technical specification
- The code itself (for structural testing)
- A business process

Test Condition

- It is a set of rules under which a tester will determine if a requirement is partially or fully satisfied
- One test condition will have multiple test cases

Test Case Terminologies (cont.)

Test Scenario

- It is an end-to-end flow of a combination of test conditions & test cases integrated in a logical sequence, covering a business processes
- This clearly states what needs to be tested
- One test condition will have multiple test cases

Test Procedure (Test Steps)

- A detailed description of steps to execute the test

Test Data/Input

- Inputs & its combinations/variables used

Expected Output

- This is the expected output for any test case or any scenario

Actual Output

- This is the actual result which occurs after executing the test case

Test Result/Status

- Pass / Fail – If the program works as given in the specification, it is said to Pass otherwise Fail.
- Failed test cases may lead to code rework

Other Terminologies

Test Suite – A set of individual test cases/scenarios that are executed as a package, in a particular sequence and to test a particular aspect

- E.g. Test Suite for a GUI or Test Suite for functionality

Test Cycle – A test cycle consists of a series of test suites which comprises a complete execution set from the initial setup to the test environment through reporting and clean up.

- E.g. Integration test cycle / regression test cycle

Test Suite – Test suite is set of all test cases. Suites are usually related by the area of the application they exercise or by their priority or content.
For E.g. When you Login to the screen, some functionalities like validating user name, password with different invalid inputs can act as Test suites.
E.g. In case of ATM machine, deposit, withdraw, balance check are separate test suites that carry out different test cases

Test Cycle – It's a combination of series of test suites.
For E.g. The Test Cycle for the same application is combination of multiple Test Suites like, Functional validations, Database validations, GUI validations, etc. form the Test Cycle.
E.g. Combination of all test suites like deposit, withdraw, balance check in case of ATM machine, forms a complete cycle

A good Test Case

Has a high probability of detecting error(s)
Test cases help us discover information
Maximize bug count
Help managers make ship / no-ship decisions
Minimize technical support costs
Assess conformance to specification
Verify correctness of the product
Minimize safety-related lawsuit risk
Find safe scenarios for use of the product
Assure quality

Features of a good Test Case:

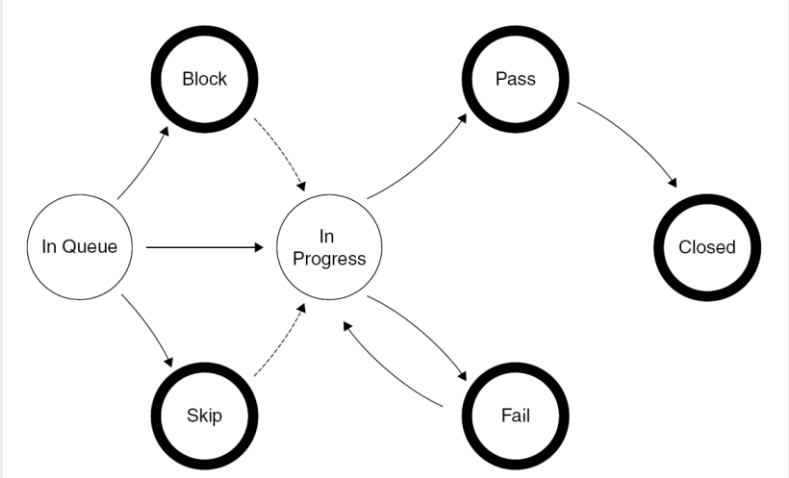
Detecting defects. This is the classic objective of testing. A test is run in order to trigger failures that expose defects. Generally, we look for defects in all interesting parts of the product.

Maximize bug count. The distinction between this and “find defects” is that total number of bugs is more important than coverage. We might focus narrowly, on only a few high-risk features, if this is the way to find the most bugs in the time available.

Help managers make ship / no-ship decisions. Managers are typically concerned with risk in the field. They want to know about coverage (maybe not the simplistic code coverage statistics, but some indicators of how much of the product has been addressed and how much is left), and how important the known problems are. Problems that appear significant on paper but will not lead to customer dissatisfaction are probably not relevant to the ship decision.

Minimize technical support costs. Working in conjunction with a technical support or help desk group, the test team identifies the issues that lead to calls for support. These are often peripherally related to the product under test--for example, getting the product to work with a specific printer or to import data successfully from a third party database might prevent more calls than a low-frequency, data-corrupting crash.

Test Case Lifecycle



Test Case Design Techniques

As Exhaustive testing is impractical, test case design techniques will help us to select test cases more intelligently

What is a Test Design Technique?

- A procedure for selecting or designing tests
- Based on a structural or functional model of the software
- Successful at finding faults
- Best' practice
- A way of deriving good test cases
- way of objectively measuring a test effort

Advantages of Test Design Techniques:

- Different people: similar probability find faults
- Effective testing: find more faults
- Efficient testing: find faults with less effort

A testing technique helps us select a good set of tests from the total number of all possible tests for a given system.
Each technique provides a set of rules or guidelines for the tester to follow in identifying test conditions and test cases.

Test Case Design Techniques (Cont.)

Test cases are designed based on the following techniques

- Static (non - execution)
  - Examination of documentation, source code listings, etc.
- Specification-based - Black Box testing techniques
  - Boundary value analysis, Equivalence partitioning, decision table
- Structure – based – White Box testing techniques
  - Code coverage, decision coverage, statement coverage
- Experience based techniques
  - Exploratory testing, fault attack, error guessing

Experience based Technique

In experience-based techniques, people's knowledge, skills and background are of prime importance to the test conditions and test cases.
The experience of both technical and business people is required, as they bring different perspectives to the test analysis and design process. Because of the previous experience with similar systems, they may have an idea as what could go wrong, which is very useful for testing.
This technique is used for low-risk systems.

Exploratory testing
It is a hands-on approach in which testers are involved in minimum planning and maximum test execution.
The test design and test execution activities are performed in parallel typically without formally documenting the test conditions, test cases or test scripts.

What is test data?

Test Data

- An application is built for a business purpose. We input data and there is a corresponding output. While an application is being tested we need to use dummy data to simulate the business workflows. This is called test data.
- A test scenario will always have an associated test data.Tester may provide test data at the time of executing the test cases or application may pick the required input data from the predefined data locations.
- The test data may be any kind of input to application, any kind of file that is loaded by the application or entries read from the database tables. It may be in any format like xml test data, stand alone variables, SQL test data etc.

If you are testing with bad or unstable data, how can you be sure your test results are accurate!!!

Properties of Good Test Data

Realistic – accurate in context of real life

- E.g. Age of a student giving graduation exam is at least 18

Practically valid – data related to business logic

- E.g. Age of a student giving graduation exam is at least 18 says that 60 years is also valid input but practically the age of a graduate student cannot be 60

Cover varied scenarios

- E.g. Don't just consider the scenario of only regular students but also consider the irregular students, also the students who are giving a re-attempt, etc.

Exceptional data

- E.g. There may be few students who are physically handicapped must also be considered for attempting the exam

Test Data team

Test Data team should have Data Coordinator and team members. Test data teams are structured in various different ways.

- Dedicated test data team
- Development team as a test data team
- QA team as a test data team

Data Coordinator's role and responsibility - Data coordinator will be the point of contact between the main stakeholders. He will be responsible for gathering all data requirements.

- Documentation of knowledge of interfaces and test data, mentoring and advising test team on data use, and support on the end-to-end flow;
- Data Coordinator will be responsible of the data prioritization, according to the timelines fixed by data team for executing and delivering the requested data.

Self explanatory

Test Data team (Cont.)

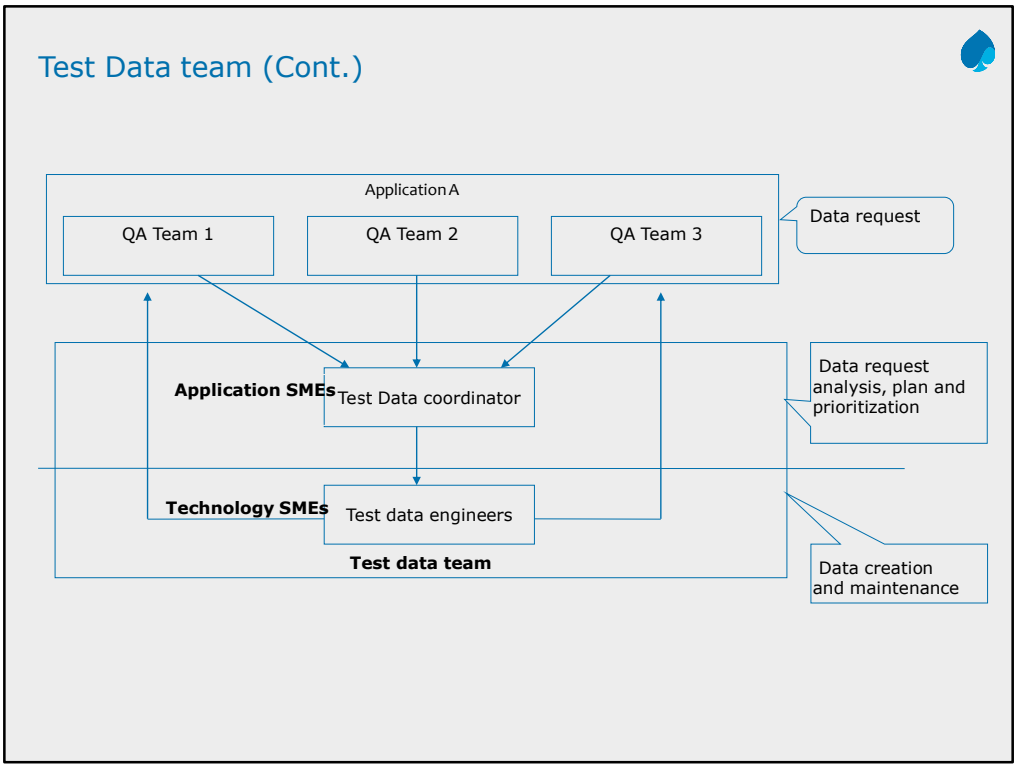
- Participation in test case planning, collection and providing data to support test cases for development and execution
- Providing help in handling, managing and manipulating test data
- Data Coordinator must ensure the correct application of masking rules, since each test case can have a set of static data necessary for execution

Test Data Engineer's role and responsibility – Test data engineer will be responsible to create the data as per the requirement . He should be doing following activities during data creation

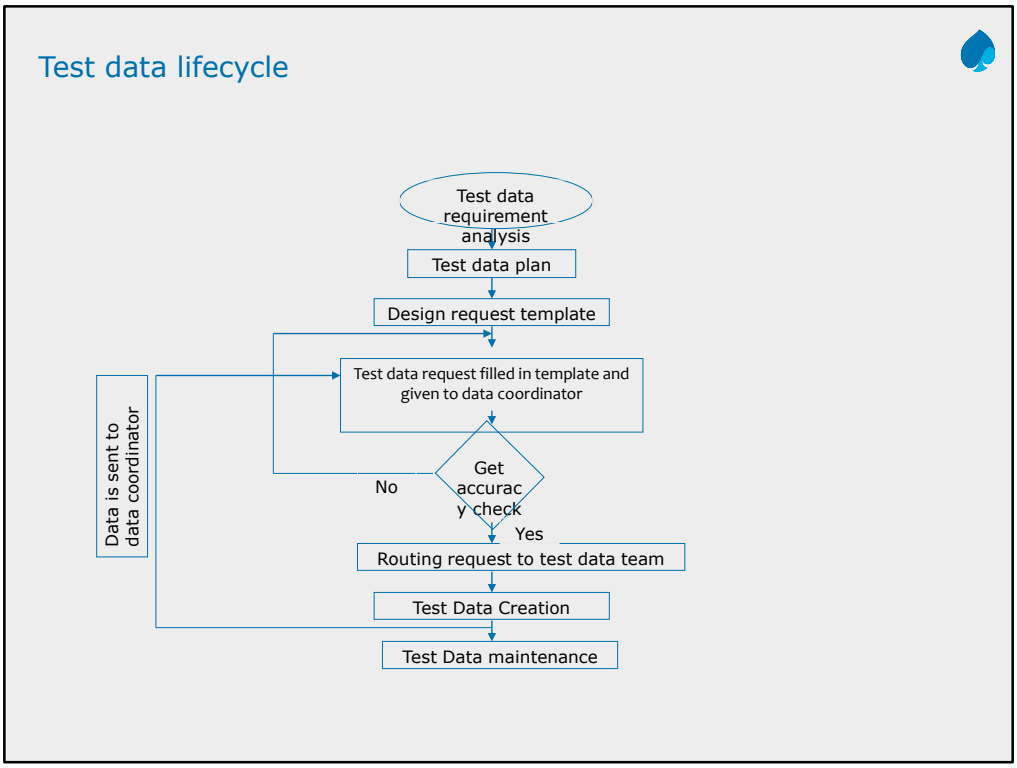
- Understand the Requirements
- Understand the DB and Table structures of the applications in case data generation for database testing
- Understand the volume of data required
- Automate the process of data generation if volume is huge

Self explanatory

Test Data team (Cont.)



Self explanatory



The way we have SDLC and STLC for software, the same way test data has to follow a defined path.

It having similar phases as STLC:

Requirement
Plan
Creation
Usage and maintenance

Requirement and Planning

Test data requirement analysis

- Test data analysis is the most important part of test data management. While preparing the test cases and test scripts, the test analyst needs to understand the exact data requirement and needs to parameterize the same
- Parameter-zing the test data requirement for the test cases would reduce the effort in understanding the test data requirement during the data mapping phase
- Test data might be converted data from the legacy system. At this point initial test data analysis and parameterization would help to identify those data requirements that can be picked from legacy system and which need to create from scratch

Once test case design is over test data requirement should also be completed.

Traceability matrix will be generated to ensure complete coverage of requirements.

Parameterzing means a value or a symbolic reference to a value.
Give some value for each data requirement. So If sample is available then its easy to understand the exact data requirements.

Requirement and Planning (Cont.)

Test data planning – Test data planning includes following:

- Test data requirement template finalization
- Defining the request initiation and completion process
- Data organization
- Data creation tools and technologies
- Test data state control mechanism
- Data maintenance planning

Planning - > A standard template will always be helpful during requirement gathering as all the users have the common understanding of the fields in template.
By looking at the requirements need to plan if there is any tool is require to generate the data. For example any kind of database testing will be required millions of records that are not possible to create manually.
Once data is created it might required for subsequent iterations or releases, in that case data need to maintain.
For maintenance it's required to know the data state after the execution and the required state for next iteration. So proper data state control mechanism should be in place.

Request Process

The communication between data coordinator and QA Testing Team must follow certain standards in terms of requests template. The data coordinator will act as a single entry point for the QA Team and will be able to retrieve data for testing.

- Data request Template
  - The Testing Team will address their requests to the data coordinator through a standard template
  - The template includes fields to fulfil with detailed information's for each kind of requested data. The tester has to describe very precisely the needs of required data for execution of test cases to ensure quick and consistent delivery of data from the data team
  - The fields integrated in the request template are grouped for each kind of data
  - The QA Testers have to fill the template and when it's possible add a sample of required data
  - The data coordinator will be in charge of the data request template from QA team

Request Process (Cont.)

Request Accuracy check

- After the receipt of the request, a manual accuracy check is executed by the data Coordinator on the following points:
  - Validate the template used by the users whether it's standard one or not
  - All mandatory fields are filled in and filled in properly
  - Any duplicated request or incomplete request templates
  - Requests of data already existing (extracted) in Data Base
  - The data coordinator will provide the reason for having rejected the template and then acknowledge the requestor by email
  - In case the request file from the tester passes the Accuracy Check, then the data coordinator will move that request to data creation team

Give example of existing project.

Test Data Creation Techniques

There are two ways to create the test data.

- Data from scratch – Any new development project, there will be no production data to mimic. So there is a need of data creation from scratch. Though this approach is free of all the privacy issues, still it is very difficult to actually come up with test data easily that actually mimics production environment
- While designing the test data following categories need to consider:
  - **No data:** In case of no input data proper error message should pop up.
  - **Valid data set:** Data is created to check whether required behavior is exhibit if data given is a valid data
  - **Invalid data set:** Check for negative values
  - **Illegal data format:** Make one data set of illegal data format. System should not accept data in invalid or illegal format. Also check proper error messages are generated

An application accepting integer values (that is whole number values) between -10,000 to +10,000 can be expected to be able to handle negative integers, zero and positive integers. Therefore, the set of input values can be divided into three partitions:
From -10,000 to -1, 0 and from 1 to 10,000

Test Data Creation Techniques (Cont.)

Boundary Condition data set: Data set containing out of range data. Identify application boundary cases and prepare data set that will cover lower as well as upper boundary conditions

Equivalent partition: Set of data to an input condition that give the same result when executing a program and partitioning them from another equivalent set of data for the same condition

Decision Tree: This model is used to visually represent the paths from input to all possible outputs of a given system. Data is created for each node and corresponding branches

State transition: State-Transition testing is identifying the states, events, actions, and transitions that should be tested. It also define how a system interacts with the outside world, the events it processes

Test Data From Production Data

Test data from production data

- The typical approach for data creation is using old production data or by transforming production data so as to replace actual values with equivalent values
- Production data can't be used as it is. Most of the time it's a sensitive information like credit card number etc. So masking is done over the production data to protect sensitive information
- After sanitization, the database remains perfectly usable - the look-and-feel is preserved - but the information content is secure

For example any search engine project 'statistics testing'. To check history of user searches and clicks on advertiser campaigns large data was processed for several years which is practically impossible to manipulate manually for several dates spread over many years. So there is no other option than using live server data backup for testing.

Test Data Life Cycle - Maintenance

Data maintenance

- Data maintenance is a sizeable task, and it is be substantial fraction of overall test maintenance costs

Activities

- Data maintenance team should closely with the execution team to monitor any data related defects that are raised during the execution phase
- In case of data corruption due to defect fixes or due to major deployment, the test data management team should take the lead and analyze the extent of data corruption and start data mapping activity as a part of maintenance

Test Data Life Cycle - Maintenance (Cont.)

Other data maintenance activates include:

- Replacing non-reusable data for regression testing
- Data Manipulation
- Responding to change - database schema, code, requirements
- New test requirements

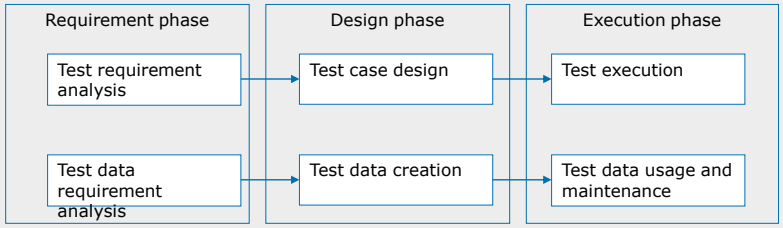
BEST PRACTICES

- Data requirement analysis as a part of preparation activity
- Parameterize the data requirement of a test case/test script
- Split the test cases as Data Reusable and Data Non-reusable
- During execution if There are data defects raise the same as a defect in a tool and assign it to the data maintenance team
- Test data maintenance should be a parallel activity along side execution since quality of data determines the quality of our testing

Data-Reusable Test case: A test case or a test script that does not change the nature of the test data that has been mapped to it. An example would be customer enquiry test cases, where customer identification number (CIN) or an account number (A/c) is mapped to the test case. While executing the test case, the data does not get corrupt. Hence this data can be reused. Such test cases are classified as Data-Reusable test cases.

Data-Non-Reusable: A test case or a test script that changes the nature of the test data that has been mapped to it comes under this category. An example would be a case where a customer is marked as deceased. Once data is modified after the execution of the test script, the same data cannot be reused across another test case.

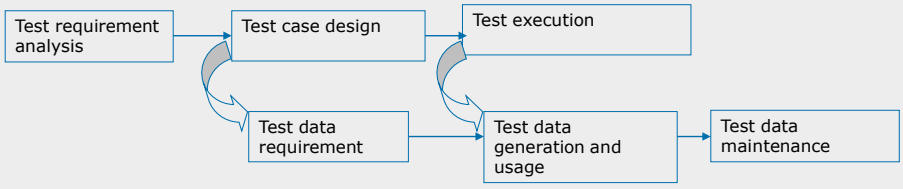
Test Data in STLC - Staggered with test case Design



+ve	-ve
•All Test data will be ready before Test Execution phase. •As Test cases are designed and signed-off, data creation will happen almost concurrently.	•There are issues such as Resource overloading, idle time due to lesser test case inputs or not having logical test cases, which may lead to load balancing challenges

Self explanatory

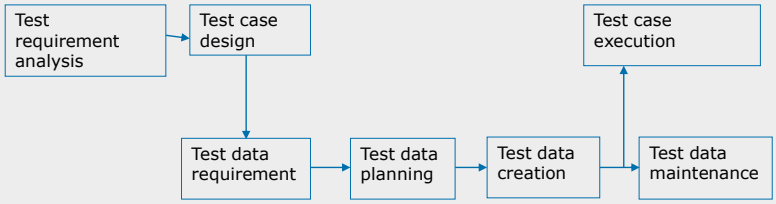
Test Data in STLC - Staggered with test case Execution



+ve	-ve
•Test data creation will happen as per Test Execution Priority •Minimizes rework as all Test Cases will be designed and signed-off prior to Test Data creation	•Test data dependencies between different areas to be tested might be overlooked •Some of the Test data creation can be redundant

Self explanatory

Test Data in STLC -Standalone phase between Test Case Design and Test Case Execution



+ve	-ve
•Test data creation happens with clear exit criteria from Test design phase and with clear entry criteria for the Test Execution phase •Data creation will be completed so There will be no or minimal resource load balancing problems	• If not planned in advance, Test Data creation activities can delay the start of Test execution activities

Self explanatory

What is Positive Testing?

Positive testing can be performed on the system by entering the valid data as input

When tester test the application from positive point of mind then it is known as positive testing

Testing aimed at showing software works

Also known as "test to pass" or "Happy path testing"

It is generally the first form of testing that a tester performs on an application

Example : Consider a scenario where you want to test an voting application which contains a simple textbox to enter age and requirement is that it should take only integers values and the value should be greater than 18.

Age: 19 Enter only integer values > 18 (Positive Testing)

Note: Equivalence Partitioning and boundary value analysis are test case design techniques used for writing positive test cases.

Advantages/Limitations of positive testing

Advantages of positive testing

- Positive testing proves that a given product and project always meets the requirements and specifications
- Positive testing ensures that the business use case is validated

Limitations of Positive testing:

- Positive tests check for only valid set of values

What is negative testing?

The purpose of Negative testing is to break the system and to verify the application response for invalid inputs

This is to test the application that does not do anything that it is not supposed to do

When tester/User test the application from negative point of mind then it is known as negative testing

Testing aimed at showing software does not work. Also known as "test to fail"

Example of Negative testing:

- In the voting application scenario, Negative testing can be performed by testing by entering alphabets characters from A to Z or from a to z. Age text box should not accept the values or it should throw an error message for these invalid data inputs.

Age:	ABC223	Enter only integer values > 18

Note:Equivalence Partitioning and boundary value analysis are test case design techniques used for writing negative test cases.

Advantages/Limitations of negative testing

Advantages of Negative Testing:

- Negative testing helps to improve the testing coverage of your software application under test
- Negative testing discovers 'hidden' errors from application under test
- Negative testing help to find more defects & improve the quality of the software application under test
- negative testing ensures that the delivered software has no flaws

Limitation of Negative Testing

- Negative tests check for only invalid set of values

Positive & Negative test scenarios

Let's take example of Positive testing scenarios:

- If the requirement is saying that password text field should accepts 5 – 15 characters and only alphanumeric characters.

Positive Test Scenarios:

- Password textbox should accept 5 characters
- Password textbox should accept up to 15 characters
- Password textbox should accepts any value in between 5-15 chars length
- Password textbox should accepts all numeric & alphabets values

Negative Test Scenarios:

- Password textbox should not accept less than 5 characters
- Password textbox should not exceeds more than 15 characters
- Password textbox should not accept special characters

What is Basic test?

Basic tests are used to test very basic functionality of software
The basic tests also verifies end to end builds
Basic test are always positive tests
Basic test can be smoke test or sanity test

Example on Basic test

Customer Relationship Management (CRM) application is business philosophy towards customers. To focus on their needs and improve customer relationships, with view to maximize customer satisfaction. So, in CRM application customer creation is basic functionality that should work. So the basic test focus is on Login and then customer creation. The basic test for this CRM application is Customer login and then customer creation with mandatory fields.

What is Alternate test?

Sometimes there maybe more than one way of performing a particular function or task with an intent to give the end user more flexibility or for general product consistency

This is called alternate testing

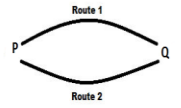
Alternate test is a kind of positive testing

In alternate path testing the test is again performed to meet its requirements but using different route than the obvious path

The test scenario would even consume the same kind of data to achieve the same result

Example on Alternate test

Alternate test



P is a starting point and Q is the end point. There are two ways to go from P to Q. Route 1 is the generally taken route and Route 2 is an alternative route. Therefore in such a case, happy path testing would be traversing from point P to Q using Route 1 and the alternate test would comprise taking Route 2 to go from P to Q. Observe that the result in both the cases is the same.

Importance of writing positive, negative, basic, alternate test while designing test cases

Approach of writing positive, negative, basic, alternate test are useful to design effective test cases which help to improve quality of software

These approach to test case design are help to improve the test case design coverage

By using these approach test cases are written for real life scenarios. It ensures real life scenarios are tested before moving software live

By designing positive and negative test cases ensures that the application works as per the requirements and specifications

By executing effective test cases, helps to find more defects before releasing software, so it builds confidence in system

Best practices for test case maintenance

Have Approved Test Case template in place

Identify the location of Test Cases storage with good access control

Have Test Case Review & Approval SOP in place

Appropriate Training for Testers for Test Case Authoring / Reviews / Executions / Maintenance

Test Cases attributes can be discussed and agreed upon:

- Should it contain Navigational OR click-by-click Test Steps
- Should it contain Test Data set up steps within Test Case or it should be done separately
- Test Case modification protocol
- Reusable components development

It's really important to think about how you structure and divide your test cases to make them as reusable as possible.

For example, you need to write test cases to test to flow say

Login->View cosmetic product->Select cosmatic product->Checkout.

You should write a short test case for each function, rather than writing a huge test case that tests the entire flow. In this way way, you can reuse the same test cases even if the flow changes, merely re-tying them.

Best practices for test case maintenance

Audit Trail for every update in Test Case

Good management of Impact assessments with every update in requirement(s) w.r.t Test Cases coverage

Better management of Trace Matrix with “every” update

Maintenance of record of Executed Test Cases and Defects for reference of updates

It is advisable to store test cases in version control tool so that any subsequent changes can be tracked easily

Use test case creation and maintenance tools. One such tool is Quality Center from HP

Summary

In this lesson, you have learnt:

- The test case techniques discussed so far need to be combined to form overall strategy
- Each technique contributes a set of useful test cases, but none of them by itself contributes a thorough set of test cases

Review Question

Question 1: _____ testing can discover dead codes

Question 2: The objective of walkthrough is to find errors but not solutions

- Option: True / False

Question 3: For calculating cyclomatic complexity, flow graph is mapped into corresponding flow chart

- Option: True / False

Question 4: How many minimum test cases required to test a simple loop?

Question 5: Incorrect form of logic coverage is :

- Statement coverage
- Pole coverage
- Condition coverage
- Path coverage

Review Question

Question 6: One test condition will have _____ test cases.

Question 7: For Agile development model conventional testing approach is followed.

- Option: True / False

Question 8: A test case is a set of _____, _____, and _____ developed for a particular objective.

Question 9: An input field takes the year of birth between 1900 and 2004. State the boundary values for testing this field.

- 0, 1900,2004,2005
- 1900, 2004
- 1899, 1900, 2004, 2005
- 1899, 1900, 1901, 2003, 2004, 2005

Review Question: Match the Following

1. Code coverage	A. Flow graph
2. Interface errors	B. Loop testing
3. Code complexity	C. Black box testing
	D. Flow chart
	E. Condition testing
	F. White box testing