Software Testing

Software Fault:- A static defect in the software
Software Failure:- External incorrect behavior with respect to requirements
Software Error:- An incorrect internal state that is manifestation of some fault

Testing Approaches

- Testing
- Verification
- Inspections
- Formal proofs of correctness

1) <u>Testing</u>:- Testing on values from the input domain and creating an output domain. Not that effective since there are no false positives. Incomplete
2) <u>Verification</u>:- Testing on all possible values in the input domain. This has two caveats:-
   1) Can have false positives
   2) Compromised accuracy due to using unintended inputs for testing
3) <u>Inspection</u>:-
   - Reviews of the code
   - Walk throughs, manual inspection (dry running)

- Done in groups

Pros:- Thorough, Systematic

Cons:- Informal, Subjective

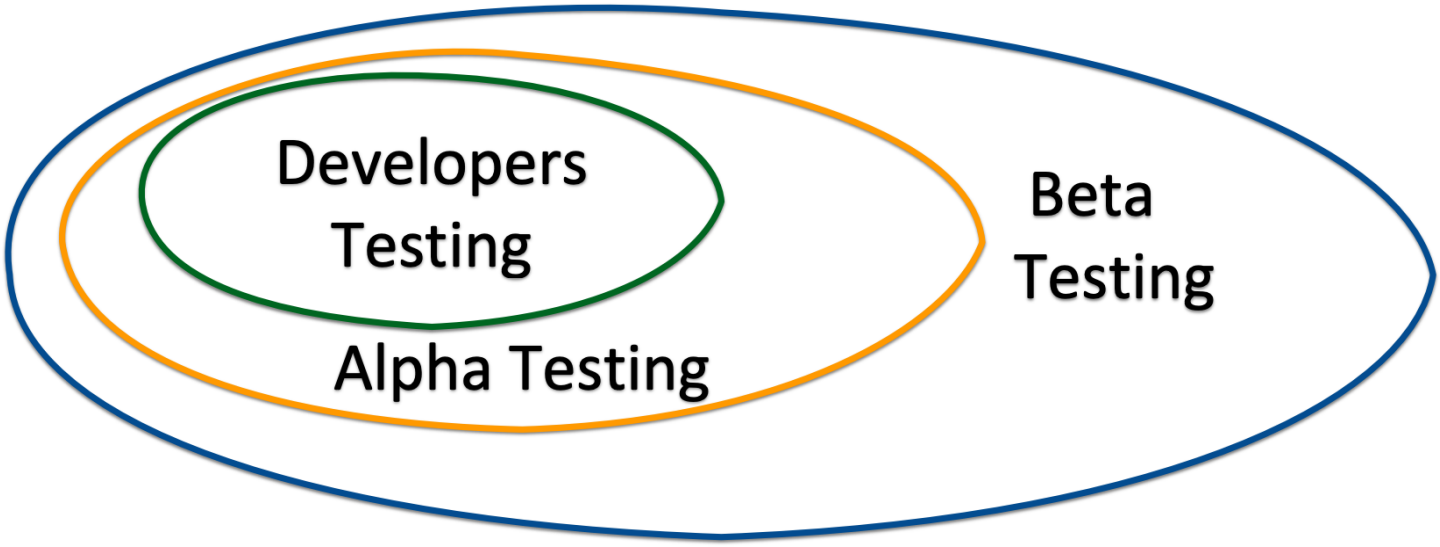
4) Formal Verification:-

Formal proof of correction

Pros:- Sound guarentees

Cons:- Complex, expensive

Testing Methodology



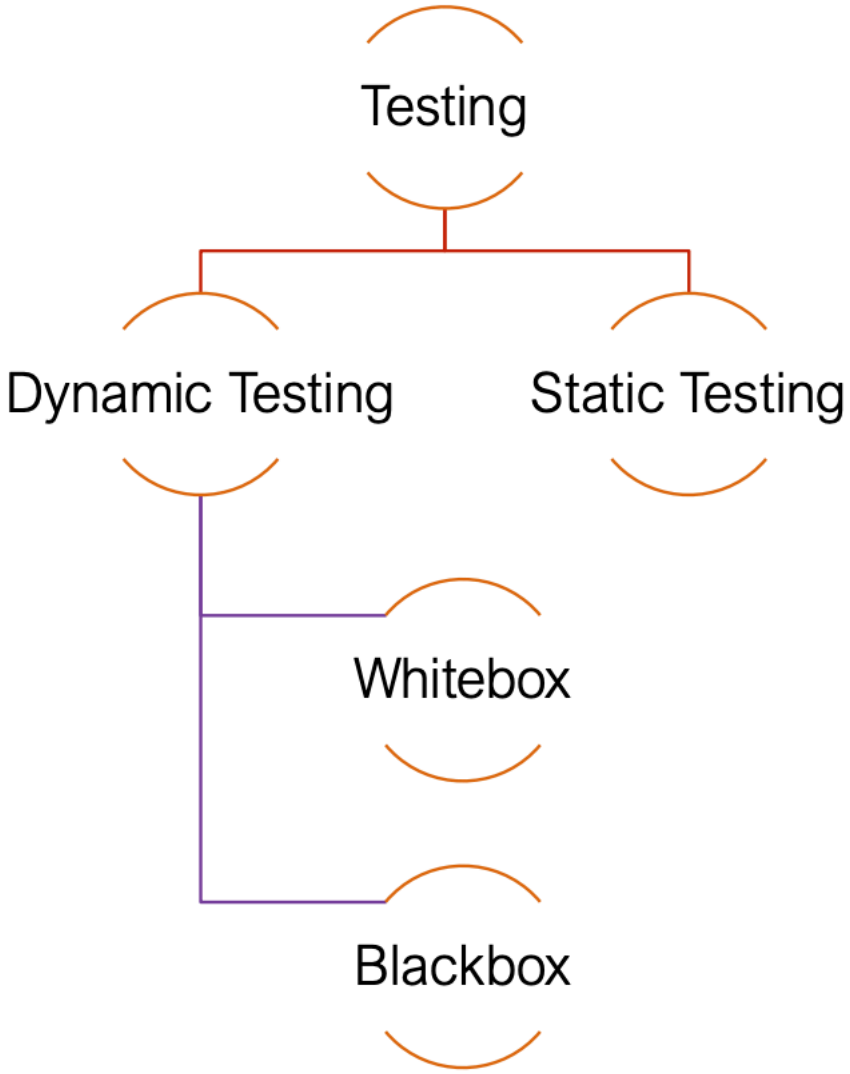
1) Alpha Testing :- Done by people in the organization with a main focus on functional requirements

2) Beta Testing :- Done by external users with a main focus on non-functional requirements

3) Developers Testing :-

- Unit Testing :- Each developer checks his/her function respectively
- Integration Testing :- Combine multiple functions/units together to see if they work well
- System Testing :- Test the entire system with all the different components to verify both functional and non-functional reqs
- Acceptance Testing :- Done by clients to see if the software performs as expected and fulfills all the requirements
- Regression Testing :- Testing done after refactoring the program after deployment. Easier for maintainable code

Software Testing Types


Testing
Dynamic Testing
Static Testing
Whitebox
Blackbox

Dynamic Technique -> Whitebox
Optimistic Approximation -> Blackbox

1) Blackbox:- Based of SRS document /client specification and thus covers only specific behavior. Can not reveal errors due to implementation details
2) Whitebox:- Based on the code provided only and tries to cover that as much as possible. Can not reveal errors due to missing paths

Writing Test Cases

1) Test Oracle:- Set of test cases representing the desired behavior of the program on a given input

SL.No	Test Case	Excepted Result	Test Result
1	On the click of ADD button	At first user have to fill all fields with proper data , if any Error like entering text data instead of number or entering number instead of text..is found then it gives proper message otherwise Adds Record To the Database	successful
2.	On the Click of DELETE Button	This deletes the details of student by using Register no.	Successful
3.	On the Click of UPDATE Button	Modified records are Updated in database by clicking UPDATE button.	Successful
4.	On the Click of SEARCH Button	Displays the Details of book for entered Register no. Otherwise gives proper Error message.	Successful
5.	On the Click of CLEAR	Clears all fields	Successful

2) Test Log:- Similar to the tracebility matrix and helps to keep track of each test case alongside the expected result

Test	Test	Test Steps	Test Data	Expected	Actual	Pass/Fail

Case ID	Scenario			Results	Results	
TU01	Check Customer Login with valid Data	1. Go to site http://demo.guru99.com 2. Enter UserId 3. Enter Password 4. Click Submit	Userid = guru99 Password = pass99	User should Login into an application	As Expected	Pass
TU02	Check Customer Login with invalid Data	1. Go to site http://demo.guru99.com 2. Enter UserId 3. Enter Password 4. Click Submit	Userid = guru99 Password = glass99	User should not Login into an application	As Expected	Pass

Black Box Testing

Types of errors regarding functional reqs of software:-

1) Incorrect or missing functions
2) Interface Errors
3) Initialization & Termination errors

Equivalence Class Testing

Equivalence Classes:- Partitions of the input set in which input data has the same effect on the program

Completness:- Covers entire input set

Disjoint:- Avoids redundancy

Test Cases:- One element from each equivalence class

Equivalence Class Partioning:-

1) Partition input domain in equivalence classes
2) Equivalence class represents set of data that will be treated equally
3) Use specification document to list all characteristics

of input domain

4) Need to identify those classes that will cause module to respond differently from other classes

Equivalence Class Types

1) Valid
2) Invalid → Negate characteristics of valid classes and should cause module to do something different

Defining Equivalence Classes

1) Input condition is a range:- One valid two invalid
2) Input condition requires a specific value:- One valid and two invalid
3) Input condition is a boolean:- One valid and one invalid class is defined

Equivalence Class Types

Assuming 3 input variables from classes A, B, C

$A: A1 \cup A2 \cup A3, \ldots Am$

$B: B1 \cup B2 \cup B3, \ldots Bm$

$C: C1 \cup C2 \cup C3, \ldots Cm$

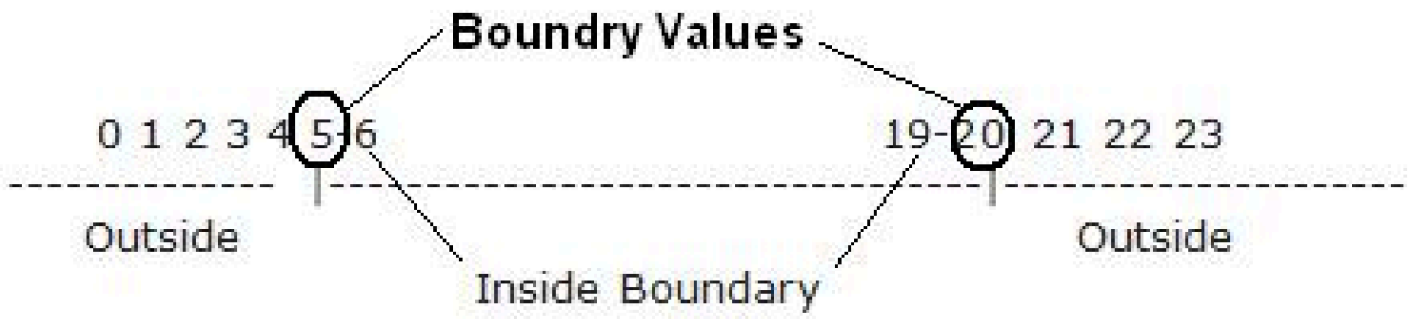
Weak Equilance Class Testing:- Choosing one variable from each equilance class for coverage

# of Test Cases = $\max(|A|, |B|, |C|)$

Strong Equilance Class Testing:- Test all interactions of all equivalence classes

Test Cases = max (|A| x |B| x |C|)

Boundary-Value Analysis



ECP selects one istance from each class but BVA requires selection of one or more elements such that we test boundaries of each equilance class
Test cases are also derived by looking at the output space

Example:-

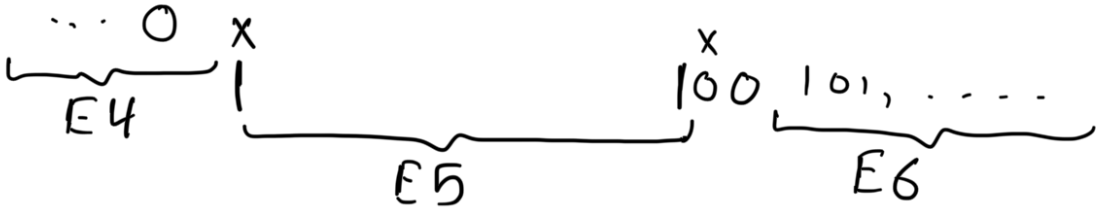
Function findPrice() has two parameters: an item code must be in the range 99..999 and quantity in the range 1..100

- Equivalence classes for code:
  - E1: Values less than 99
  - E2: Values in the range
  - E3: Values greater than 999
- Equivalence classes for qty:
  - E4: Values less than 1.
  - E5: Values in the range.
  - E6: Values greater than 100

Code = 127
Quantity = 3 → Price 87 euro

Code:-

Qty:-



X represents boundary values

Cause-Effect Table

Used when we have cause-effect relationship b/w

entities

Decision table is a tabular representation of inputs versus rules/cases

Example:-

Upload .jpg file with size not more than 32KB and resolution 137 * 177

File Format (T/F)	T	T	T	T	F	F	F	F
File Size (T/F)	T	F	F	T	T	F	F	T
File Resolution (T/F)	T	T	F	F	T	T	F	F
Output	F	F	F	F	F	F	F	F

If system behavior is different for input, not for some range of inputs then DT is better

State Transition Testing

Exposes effects triggered when moving from one program state to the other

States that the software may occupy

Transitions from one state to another

Events that cause a transition

Actions that result from a transition

Testing Strategies

1) Negative / Fuzzy Testing:-
Basically provide invalid / unexpected code to the program and monitor for exception such as crashes or build failures

2) Null Case Testing:-
Exposes defects triggered by no / missing data

3) Integration Testing:-
Individual software components are combined and tested as a group
Done after Unit Testing and before System Testing
Takes unit tested modules, groups them up, defines integration test cases and outputs the integrated system that is ready for system testing

I & T Types:-
1) Big Bang Integration
2) Top - Down
3) Bottom - Down
4) Sandwich Integration

} Incremental

1) Big Bang:- Basically all / most of the modules are combined together to form software or its major component for testing
Pros:- Saves time, no need for middle drives
Cons:- Difficult to debug, need to wait for dev to

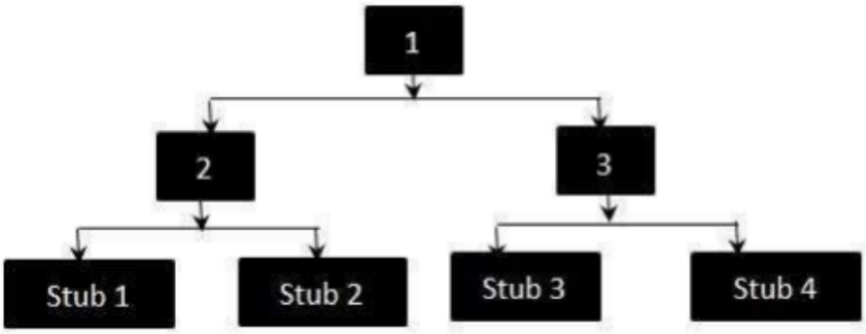
complete

2) Top-Down :- Top integrated modules are tested and the branch of the module is tested step by step till the end

Can go in a DFS or BFS fashion

Stubs :- Temporary replacement for called modules which do the same thing as the finished product

Drivers :- Simple programs designed for testing calls to lower layers



3) Bottom-Up :-

- Low level components are combined into clusters
- If there is no main module a driver is written to coordinate input and output
- After cluster is tested the driver is removed and clusters are combined while moving up

4) Sandwich Testing :-

- Combines top down and bottom up approaches
- Good for large projects having a number of

sub-projects

5) Load Testing :-

- Putting demand on a system and measuring its response
- Helps to identify maximum operating capacity and element that is causing degradation
- Relevant for multi-user systems

6) Volume Testing :-

- Testing application with a large volume of data

7) Stress Testing :-

Used to check stability of given system lentity
Basically increase load until it crashes

Differences :-

1) Volume Testing :- Large amounts of data
2) Load Testing :- Large amount of users
3) Stress Testing :- Large users and data, less time

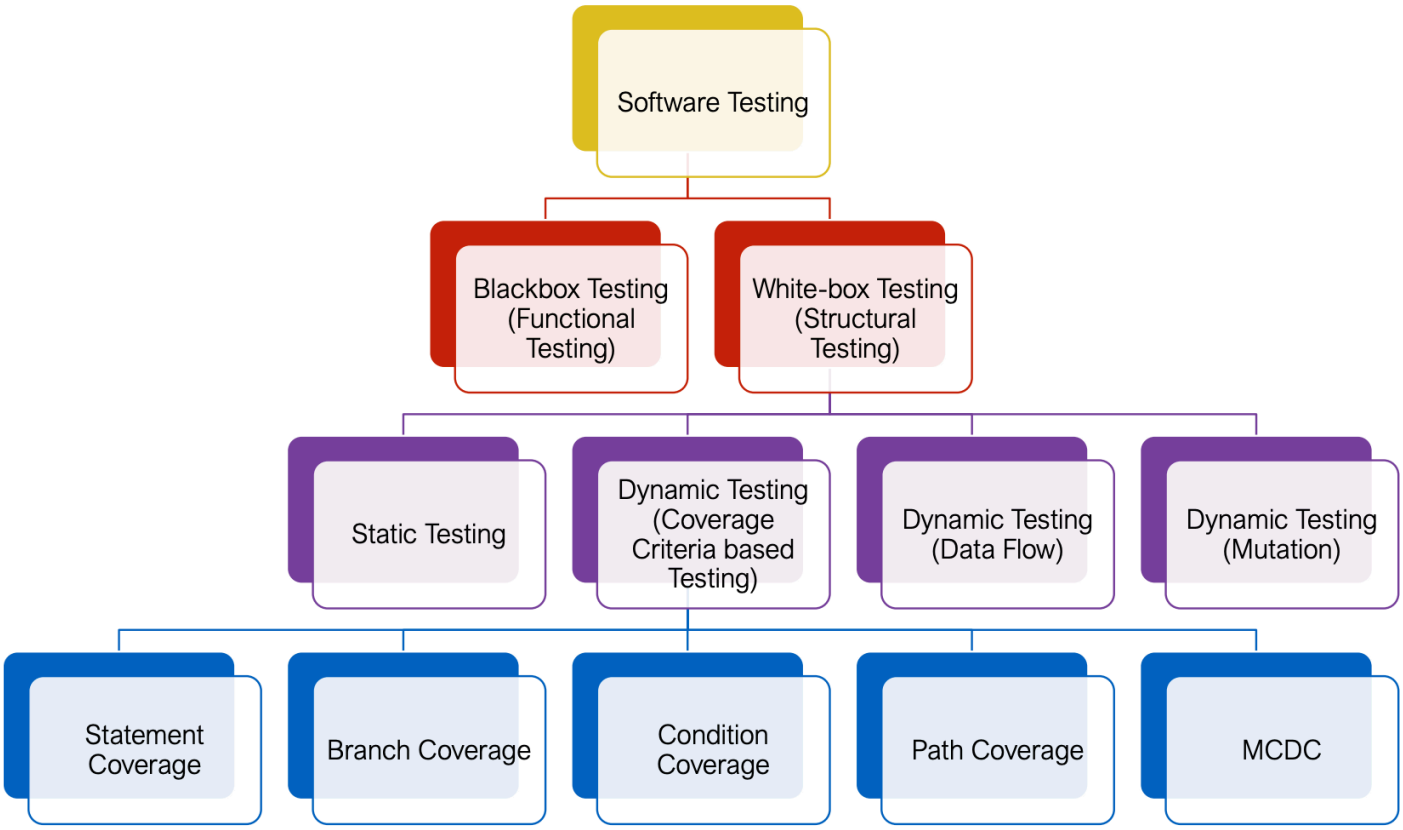
8) Regression Testing :-

Performed on already tested code when some bug in fixed or program is ported to another system

9) Progressive Testing :-

- Basically testing new code is called progressive testing
- Eventually all these become regression tests since most of the time already existing code is fixed or enhanced

White Box Testing



Testing Techniques

1) Structural Testing:-

- Requires knowledge about the internal structure of the software

Static
(No unit test cases)

Dynamic
(Unit Test Cases)
- Control Flow
- Data Flow
- Mutation

2) Static Testing:-

- Variant of Whitebox testing with no unit tests
- Relies of code walkthroughs, ispection, code reviews etc (Inspection)

Static Analysis:- Detects certain patterns in code to find potential errors due to indentation, code smells or dead code

Complexity Analysis:- Tells about hierarchy of classes and how to improve it

3) Code Coverage Based Testing:-

- Used to see what fraction of code is reachable under

already existing unit tests

Statement Coverage:- Tries to reach every line (impractical)

Condition Coverage:- Tries to reach every condition leading to a branch

Branch Coverage:- Check all branches alongside both T/F cases

Path Coverage:- Follow every path (distinct branch) through the code

→ Test cases are designed such that each statement in code is executed at least once

→ Stronger than statement coverage. Test cases are designed such that all branches are test with both T/F outcomes

→ Stronger than branch coverage. Each predicate of a composition condition is tested with both T/F outcomes.

E.g. ((c1 and c2) or c3)

c1, c2 and c3 are assigned T and F values at least once

Basically means that we need 2^n test cases in case of n components/predicates

Multiple Condition Decision Coverage (MCDC)

- Design test cases such that all outcomes of decisions as well as conditions are tested
- Basically a truth table of components/predicates

Ex:-

```
if( A >1 && B ==0)
    X = X/A
```

Two conditions so 4 test cases

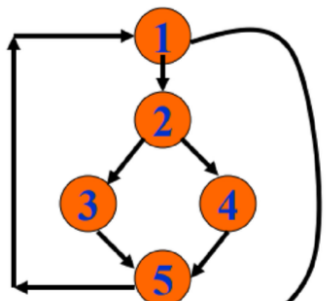
ID	A > 1	B == 0	A > 1 && B == 0
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Path Coverage

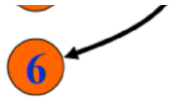
- Design test cases such that all linearly independent paths are executed at least once
- Defined in terms of Control Flow Graph (CFG) which describes the sequence in which instructions are executed

Control Flow Graph

```
int gcd(int x, int y){
1. while(x ! y){
2.   if(x > y)
3.     x = x - y;
   else
4.     y = y - x;
5. print(y);}
```



```
6. return x;
}
```



Independent Paths:- A path that introduces at least one new node that is not located in any of the other independent paths

Ex:-

1 → 6 TC:- {x = 2, y = 2}

1 → 2 → 3 → 5 → 1 → 6 TC:- {x = 2, y = 1}

1 → 2 → 4 → 5 → 1 → 6 TC:- {x = 1, y = 2}

McCabe's Cyclomatic Complexity:- Provides an upper bound on the number of linearly independent paths of a program

Given a Control Flow Graph, Cyclomatic Complexity

$= E - N + 2$

Where,

E = Edges in graph

N = Nodes

Example using above CFG,

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

$V(G) =$ Num of Predicates $+ 1 = 2 + 1 = 3$

Data Flow Testing

Explores CFG to find data-flow anomalies

Paths are explored based on data objects and we

test to ensure that these objects are initialized and all defined objects are used at least once

Data Object Categories:-

- Defined, created, initialized (d)
- Killed, undefined, released (k)
- Used (u)

Defined:- Appears in data definition, assigned a new value, dynamically allocated etc

Used:- Object is used when it is part of a Computation (C) or a Predicate (P)

→ Defined when assigned like Y = 17, used in an input statement Input(Y) or used as an OUT parameter in a function call

→ Used when appeared on RHS of assignment operator like Y = X + 17 or as an IN parameter in a function call like Y = SQRT(X) or in the predicate of a branch statement If(X>0)

Example:-

Example

- What are the definition and uses for the following program?

```
1. cin >> x >> y;
2. z = x + 2
3. if (z > y)
4.     w = x +1
```

```
5. else
6.    y = y+ 1
7. cout << x <<y <<w << z
```

1) Make a table with all the variables in program

	Define	Use
w	4	7
x	1	2, 4, 7
y	1, 6	3, 6, 7
z	2	3, 7

Imp Terms:-

- A path is definition clear which respect to a variable v if there is no redefinition to v in the same path
- A complete path is one whose initial node is the start node and final node is the exit node
- A definition-use pair (du-pair) w.r.t to variable v is a pair (d, u) such that d is a node in the flow graph at which v is defined, u is a node at which v is used and there is a def-clear path b/w d and u.