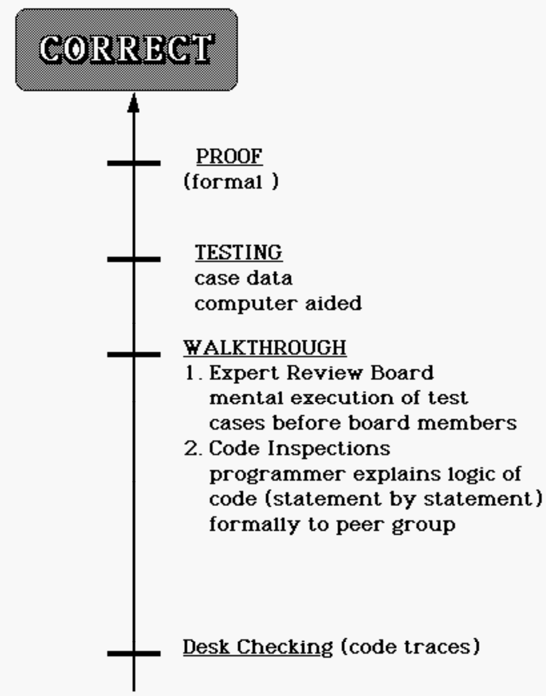
Testing

Table of Contents

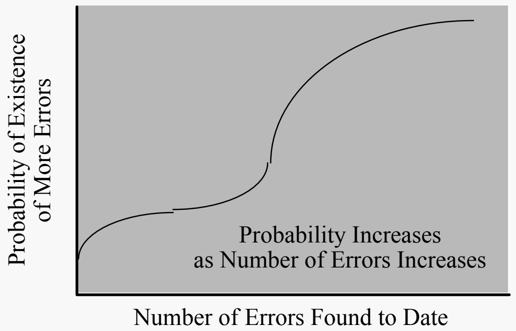
Levels of Verification

The Unreachable Goal: Correctness



Testing and Errors

Relationship between Discovered Errors and Undiscovered Errors



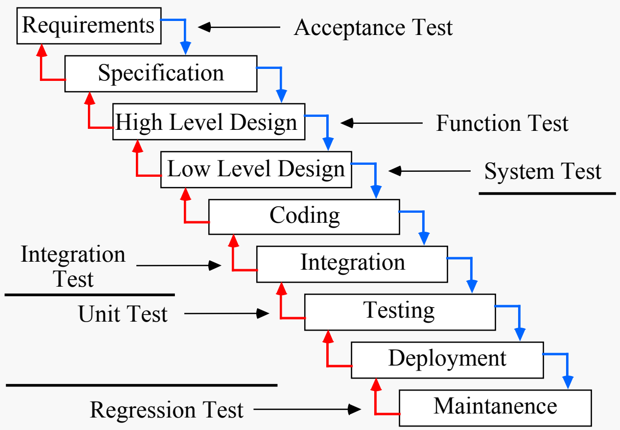
- **40-50%** of all development time is spent in the testing process
- Humans (programmers) are NOT good at testing. The process of testing admits that one has produced code with errors.
- Successful testing can be thought of as successfully finding errors and testing failure implies not discovering any errors.

> ***"Testing can establish the presence of errors, but never their absence."***
> **[Edsger Dijkstra]**

Reference:
"The Art of Software Testing", Meyers, Glenford J.,
John Wiley & Sons, 1979

Life Cycle Testing

Testing Phases



- Regression Testing involves fixing errors during testing and the re-execution of all previous passed tests.
- Unit Testing utilizes module testing techniques (white-box / black-box techniques).
- Integration Testing involves checking subsets of the system.
- Acceptance, Function and System testing is performed upon the entire system.

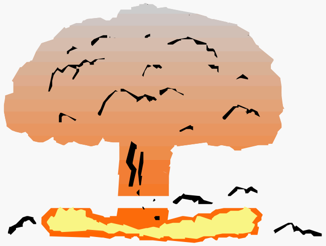
Integration Testing

Bottom-Up Testing
- Unit Test (Black & White box techniques)
- discovers errors in individual modules
- requires coding (& testing) of driver routines

Top-Down Testing
- Main module & immediate subordinate routines are tested first
- requires coding of routine stubs to simulate lower level routines
- system developed as a skeleton

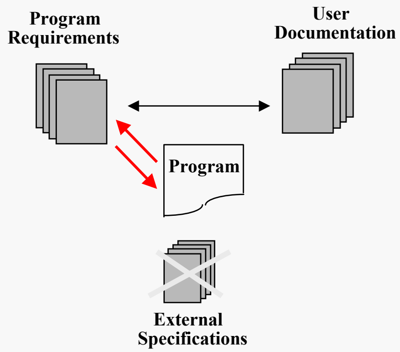
Sandwich Integration
- combination of top-down & bottom-up testing

Big Bang
- No integration testing
- modules developed alone
- All modules are connected together at once



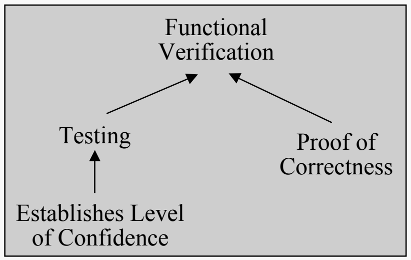
System Testing

System «-» Requirements
- Does not test the system functions
- Compares the system with its objectives, (system behavior)
- External Specification not used to compose the test cases (eliminates or reduces possible conflict of goals)
- System test cases are derived from the user documentation and requirements
- Compares user doc to program objectives
- No general system test-case-design procedure exists

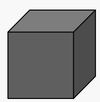


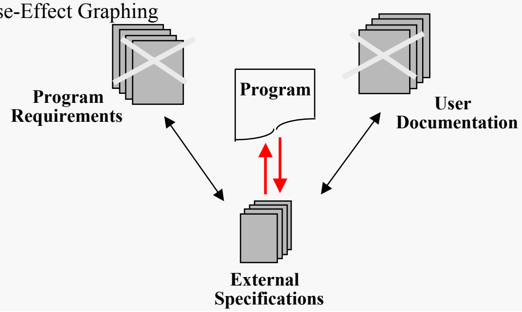
Function Testing

System «-» Specifications



- Checks that the system satisfies its external specification
- Entire system is viewed as a "**Black Box**"
- Techniques:
  - † Equivalence Partitioning
  - † Boundary-value Analysis
  - † Cause-Effect Graphing

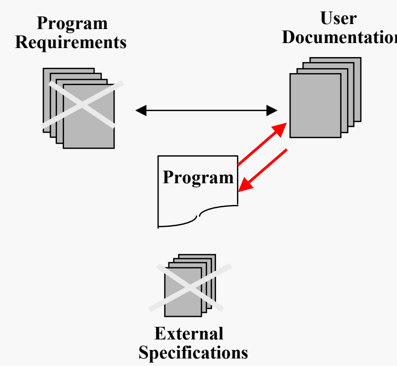




Acceptance Testing

System «-» Users

- Tests the program against the current needs of the users and its original objectives.
- Usually performed by the end user (**customer**)
- Contract may require, as part of acceptance test:
  - † performance tests (throughput, statistics collection, ...)
  - † stress tests (system limits)
- If performed by system developers may consist of α (alpha), β (beta) testing



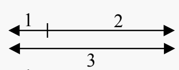
Testing Experiment

Program

- Program reads 3 integer values from a line.
- The 3 values represent the lengths of the sides of a triangle.
- The program outputs whether the triangle is equilateral, isosceles, or scalene.
- Write a set of test cases which would **adequately** test this program!

Test Cases

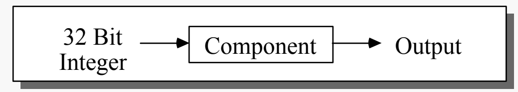
- Valid scalene triangle.
- Valid equilateral triangle.
- Valid Isosceles triangle.
- All possible permutations of Isosceles triangles (e.g. (3,3,4) (3,4,3) (4,3,3))
- One side having a zero value.
- One side having a negative value.
- Degenerate Triangle (e.g. 1-Dim Δ (1,2,3)



- All possible permutations of Degenerate Triangles (e.g. (1,2,3) (3,1,2) (1,3,2))
- Invalid Triangle (e.g. (1,2,4))
- All possible permutations of invalid triangles.
- All sides = 0.
- Non-integer values.
- Incorrect number of sides ...

Exhaustive Testing

Example



Practical Limitations

- How long will it take to try all possible inputs at a rate of one test/second?

2^{32} tests * 1 second / test

$= 2^{32}$ seconds

$= 2^{32}$ / (60 * 60 * 24 * 365) years

$> 2^{32}$ / (2^6 * 2^6 * 2^5 * 2^9) years

$= 2^{32}$ / 2^{26} years

$= 2^6$ years = 64 years

- Exhaustive Testing cannot be performed!

Testing Principles

General Heuristics

- The expected output for each test case should be defined **in advance** of the actual testing.
- The test output should be **thoroughly inspected**.
- Test cases must be written for **invalid & unexpected**, as well as valid and expected input conditions.
- Test cases should be **saved and documented** for use during the maintenance / modification phase of the life cycle.
- New test cases must be added as new errors are discovered.
- The test cases must be a **demanding exercise** of the component under test.
- Tests should be carried out by a third party independent tester, developer engineers should not privatize testing due to **conflict of interest**
- Testing must be planned as the system is being **developed**, NOT after coding.

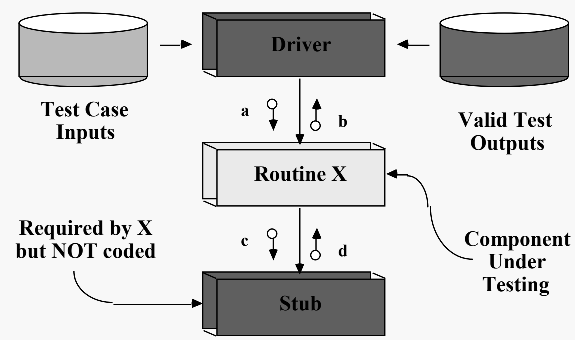
Goal of Testing

Perform testing to ensure that the probability of program/system failure due to undiscovered errors is acceptably small.

- No method (Black/White Box, etc.) can be used to detect all errors.
- Errors may exist due to a testing error instead of a program error.
- A finite number of test cases must be chosen to maximize the probability of locating errors.

Testing Mechanics

Testing components

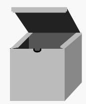


- Drivers
  - † Test harness
- Stubs
  - † Scaffold Code

White Box Testing

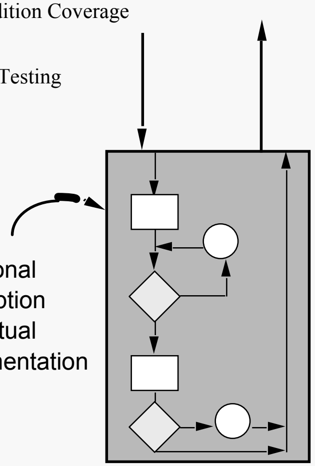
Structural Testing

- Exercise of Source code and internal data structures
- Test cases are derived from analysis of internal module logic and external module specifications
- Logic Coverage (condition/decision testing)
  - † Statement Coverage
  - † Decision Coverage
  - † Condition Coverage
  - † Decision/Condition Coverage
  - † Multiple Condition Coverage
- Path Coverage
  - † Control Flow Testing



Correct I/O relationships are verified using both :

Functional Description and actual implementation



White Box: Logic Testing

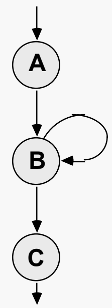
Logic Coverage

- Statement Coverage
  - † Every statement is executed at least once.
- Decision Coverage
  - † Each decision is tested for TRUE & FALSE.
  - † correctness of conditions within the decisions are NOT tested
- Condition Coverage
  - † Each condition in a decision takes on all possible outcomes at least once.
  - † Does not necessarily test all decision outcomes.
  - † Test cases do not take into account how the conditions affect the decisions.
- Decision/Condition Coverage
  - † Satisfies both decision coverage and condition coverage.
  - † Does NOT necessarily test all possible combinations of conditions in a decision.
- Multiple Condition Coverage
  - † Test all possible combinations of conditions in a decision
  - † Does not test all possible combinations of decision branches.

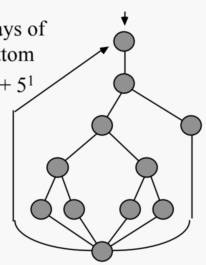
White Box: Path Testing

Control Flow Graph
- Node: sequence of statements ending in a branch
- Arc: transfer of control

Path Testing
- Exercise a program by testing all possible execution paths through the code.
- Method
  1. Enumerate the paths to be tested
  2. Find the Input Domain of each
  3. Select 1 or more test cases from domains
- Problem: Loops (∞ number of paths)
  Paths: ABC; ABBC; AB **...** BC
- Solution:
  - † Restrict loop to N iterations
  - † Select small number of paths that yield reasonable testing.

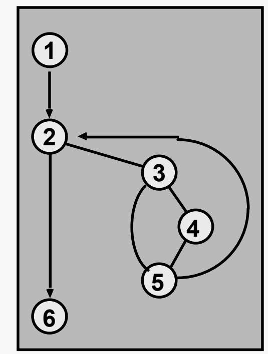


Exhaustive Path Testing (impossible)
- (analogue of exhaustive input testing)
- requires executing the total number of ways of going from the top of the graph to the bottom
- approx. 100 trillion, $10^{20} - 5^{20} + 5^{19} + \ldots + 5^{1}$ where 5 = number of unique paths
- assuming all decisions are independent of each other
- specification errors could still exist
- does not detect missing paths
- does not check data-dependent errors



Test Path Determination

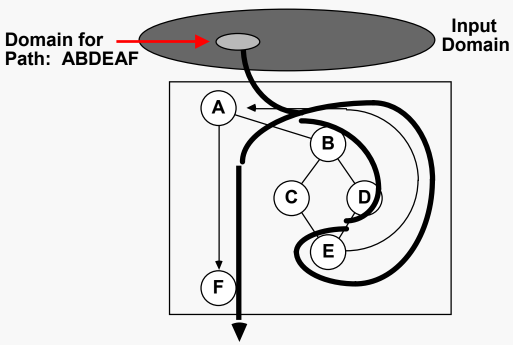
Independent Path
- any path that introduces at least one new set of processing statements (nodes), i.e. it must traverse an edge not previously covered.
- Independent Paths:
  1. 1 - 2 - 6
  2. 1 - 2 - 3 - 5 - 2 - 6
  3. 1 - 2 - 3 - 4 - 5 - 2 - 6



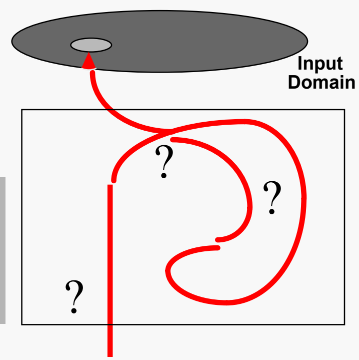
Cyclomatic Complexity
- upper bound on the number of independent paths, i.e. number of tests that must be executed in order to cover all statements.
- CC
  = edges - Nodes + 2
  = E - N + 2
  = 7 - 6 + 2 = 3
  = Predicate Nodes + 1
  = P + 1
  = 2 + 1 = 3

Path Input Domains

Input Domain Subset



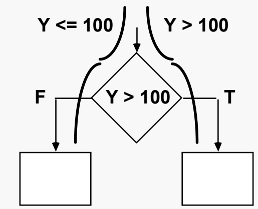
Reverse Path
Analysis



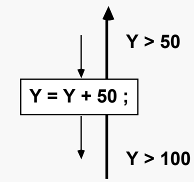
Recreate the test data by 'tracing' the path in reverse, collecting the conditions on the input variables.

Reverse Execution

Reverse execution
of a decision

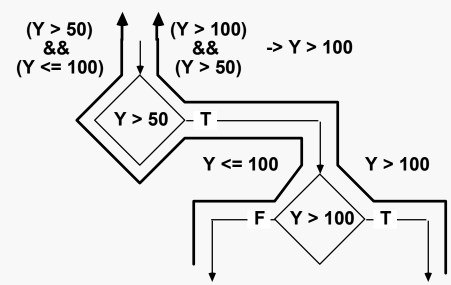


Reverse execution
of an assignment



Reverse execution of a sequence of decisions

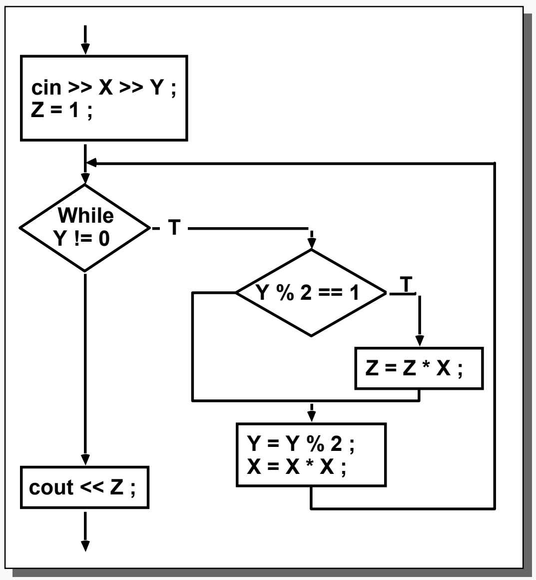
- Collected decisions are connected logically by AND.



Reverse Path Test Example

Test Component

- Computes Z = X^Y where X, Y are nonnegative integers



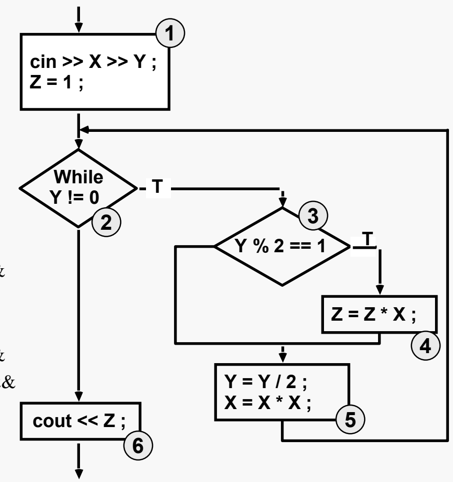
Algorithm: $$x^y = \begin{cases} \text{if y is even}: & (x^2)^{(y/2)} \\ \text{if y is odd}: & x \cdot (x^2)^{(y-1)/2} \end{cases}$$

Reverse Path Test Example (cont)

Test Path: 1 2 3 4 5 2 6

Reverse
  Path
  Execution

- **(6)**
- **(2)**
  Y = 0
- **(5)**
  Y = Y / 2
  ⇒ Y / 2 = 0
- **(4)**
- **(3)**
  Y / 2 = 0 &&
  Y % 2 = 1
- **(2)**
  Y / 2 = 0 &&
  Y % 2 = 1 &&
  Y <> 0
- **(1)**



- Test Case: Y = 1

- The input domain is bounded by the accumulated conditions.

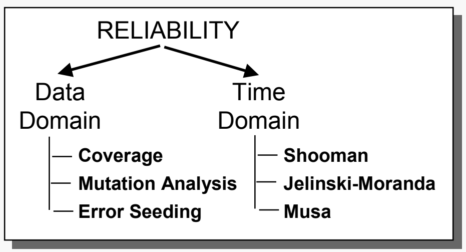
Testing Reliability

Question:

- When to stop testing?

Answer:

- When no more errors exist. Impossible to ascertain.
- (1) How reliable is the set of test cases?
  - † Data Domain
- (2) How reliable is the software being developed?
  - † Time Domain



- Time Domain Reliability

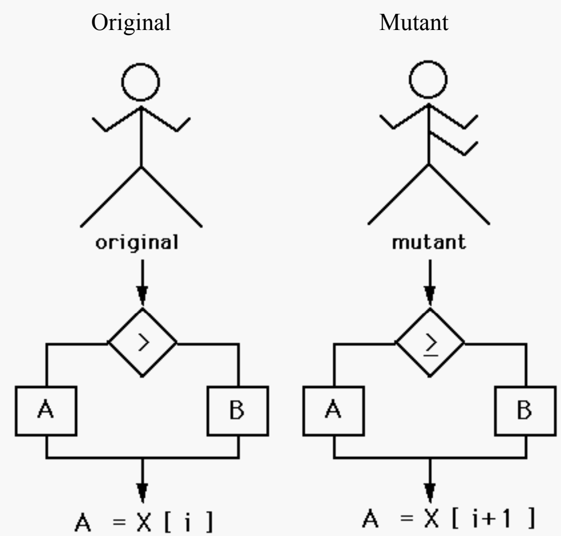
MTBF : mean time between failures
MTTF : mean time to failure
MTTR: mean time to repair

MTBF = MTTF + MTTR
Availability = MTTF / (MTTF + MTTR) * 100
Estimate Methods:

1. Predictions based on calendar time
2. Predictions based on CPU time

Mutation Analysis

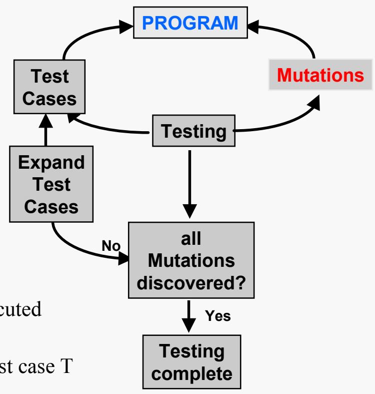
The purpose of Mutation Analysis is to test the test suite.



- Mutate Code to determine the adequacy of the test data.
- Determines whether all deliberately introduced (mutant) errors are detected by the original test cases.

Mutation Analysis Process

Mutation
Testing
Diagram



Mutation Testing Process

– 1. Program P is executed for test case T
– 2. If errors occur test case T has succeeded

Errors are corrected & retested until no errors with test case T are observed.

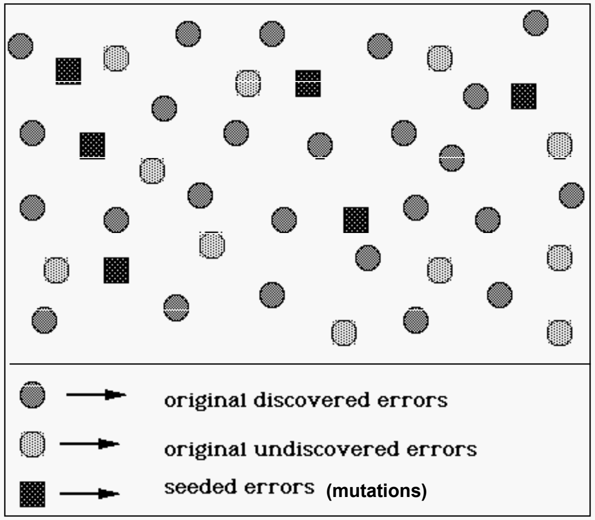
– 3. Program is Mutated P'
– 4. Mutant P' is executed for test case T

IF no errors are found {
test case T is inadequate;
further testing is required;
// ERROR SEEDING
new test cases are added & step 3 is repeated until all mutations are discovered; entire process is started again at step 1 with the new test cases

ELSE // all mutations located by tests T
T is adequate and no further testing is required.

Error Seeding

Error Scattergram Graph

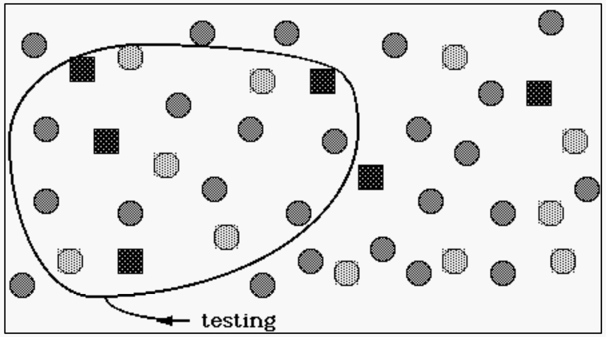


Technique

– Estimate of the number of original undiscovered errors remaining in a system.
1. Intentionally introduce (seed) errors into the source code.
2. Execute test cases upon source code.
3. Count the number of seeded errors & original errors (unseeded errors) discovered.
4. Estimate the total number of original errors

Error Seeding Process

Testing Subset



- Assume there are N undiscovered errors present in the system.
- Add S seeded errors to the system.

Test cases discover:

T_S seeded errors

T_N nonseeded (original) errors

Hypothesis:

$$\frac{T_N}{T_S} = \frac{N}{S} \quad \text{or} \quad \frac{T_S}{S} = \frac{T_N}{N}$$

$$N = S\left[\frac{T_N}{T_S}\right]$$

Test Efficiency:

$T_s/S =$ E = fraction of discovered errors