Software Engineering (CSE3004)

Software Testing

Puneet Kumar Jain

CSE Department

National Institute of Technology Rourkela

29-03-2022

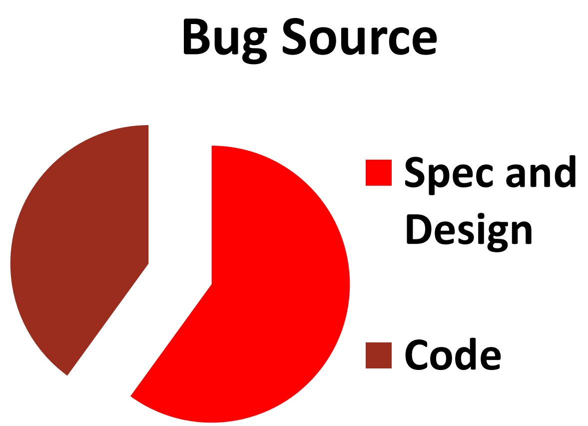
Reference

- R. Mall, Fundamentals of Software Engineering, Fifth Edition, PHI Learning Pvt Ltd., 2018.

A Few Error Facts

- Even experienced programmers make many errors:
  - Avg. 50 bugs per 1000 lines of source code

- Extensively tested software contains:
  - About 1 bug per 1000 lines of source code.
- Bug distribution:
  - 60% spec/design, 40% implementation.



Capers Jones Rule of Thumb

Each of software review, inspection, and test step will find 30% of the bugs present.

In IEEE Computer, 1996

Several independent studies [Jones],[schroeder], etc. conclude:

85% errors get removed at the end of a typical testing process.

Why not more?

All practical test techniques are basically heuristics… they help to reduce bugs… but do not guarantee complete bug removal…

Testing Facts

- Consumes the largest effort among all development activities:
  - Largest manpower among all roles
  - Implies more job opportunities
- About 50% development effort
  - But 10% of development time?
  - How?

    Parallelism
- Testing is getting more complex and sophisticated every year.
  - Larger and more complex programs
  - Newer programming paradigms
  - Newer testing techniques
  - Test automation

Basic concepts and terminologies

Mistake, Error, Fault, Bug, Failure

- IEEE std 1044, 1993 defined errors and faults as synonyms: However, **IEEE Revision of std 1044 in 2010 introduced finer distinctions:**

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution.

- An **error** is the result of a mistake committed by a developer in any of the development activities.
  - difference between Actual Output and Expected output.

- **Fault**: It is a condition that causes the software to fail to perform its required function.

Ref: https://www.360logica.com/blog/difference-between-defect-error-bug-failure-and-fault

Basic concepts and terminologies

- **BUG:** A bug is the result of a coding error. An Error found in the development environment before the product is shipped to the customer. **Bug** is terminology of Tester

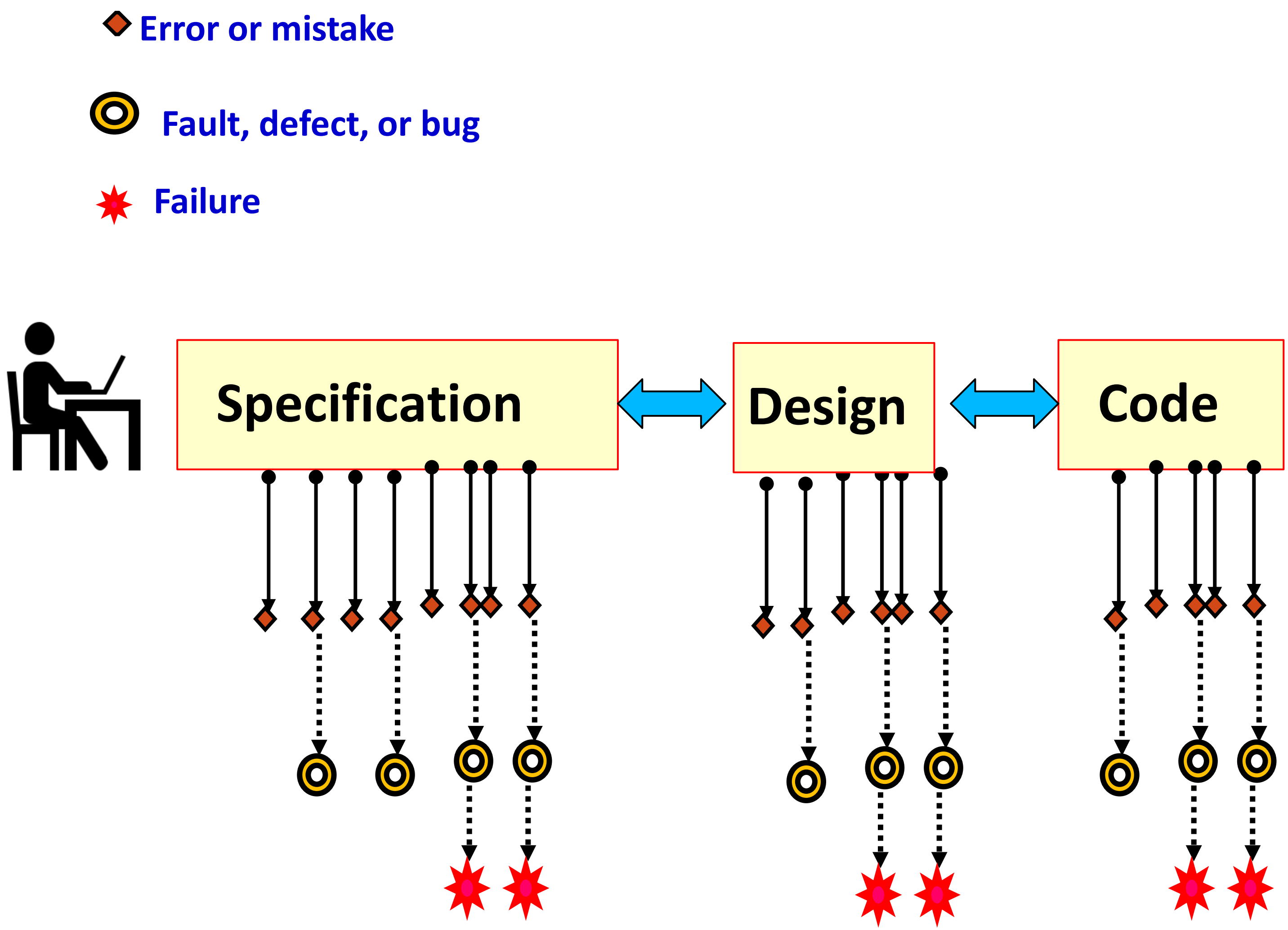
- **Failure:** a manifestation of a fault (also called defect or bug).
  - A **failure** of a program essentially denotes an incorrect behaviour exhibited by the program during its execution.

Basic concepts and terminologies

- **IEEE Definitions**
  - **Error:** Human mistake that caused fault
  - **Fault:** Discrepancy in code that causes a failure.
  - **Failure:** External behavior is incorrect

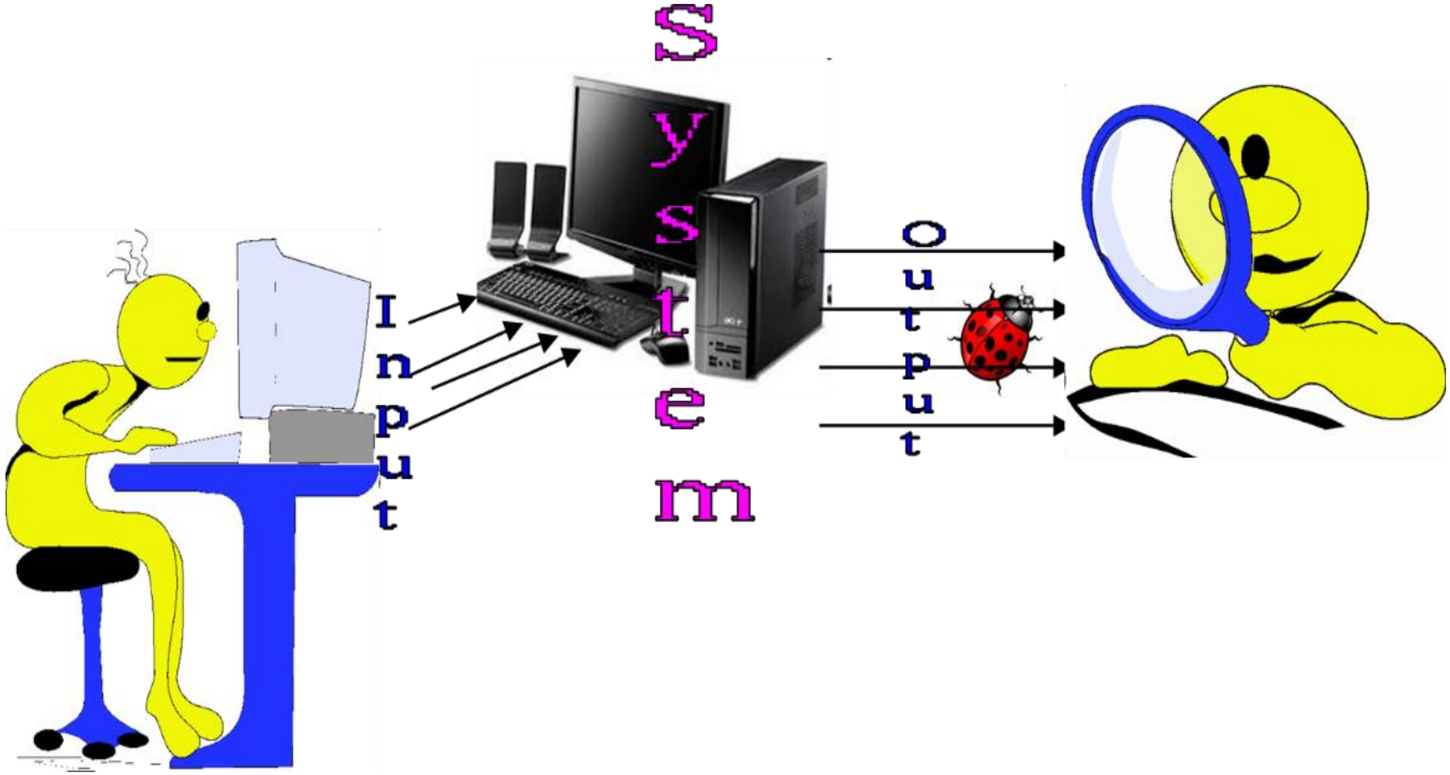
- **Note:**
  - **Error** is terminology of Developer.
  - **Bug** is terminology of Tester

Errors, Faults, Failures



Basic concepts and terminologies

- How to Test?
  - Input test data to the program.
  - Observe the output:
  - Check if the program behaved as expected.



Basic concepts and terminologies

Test Cases: A test case is a triplet **[I,S,O]**

- **I** is the data to be **input** to the system,
- **S** is the **state** of the system at which the data will be input,
- **O** is the expected **output** of the system.

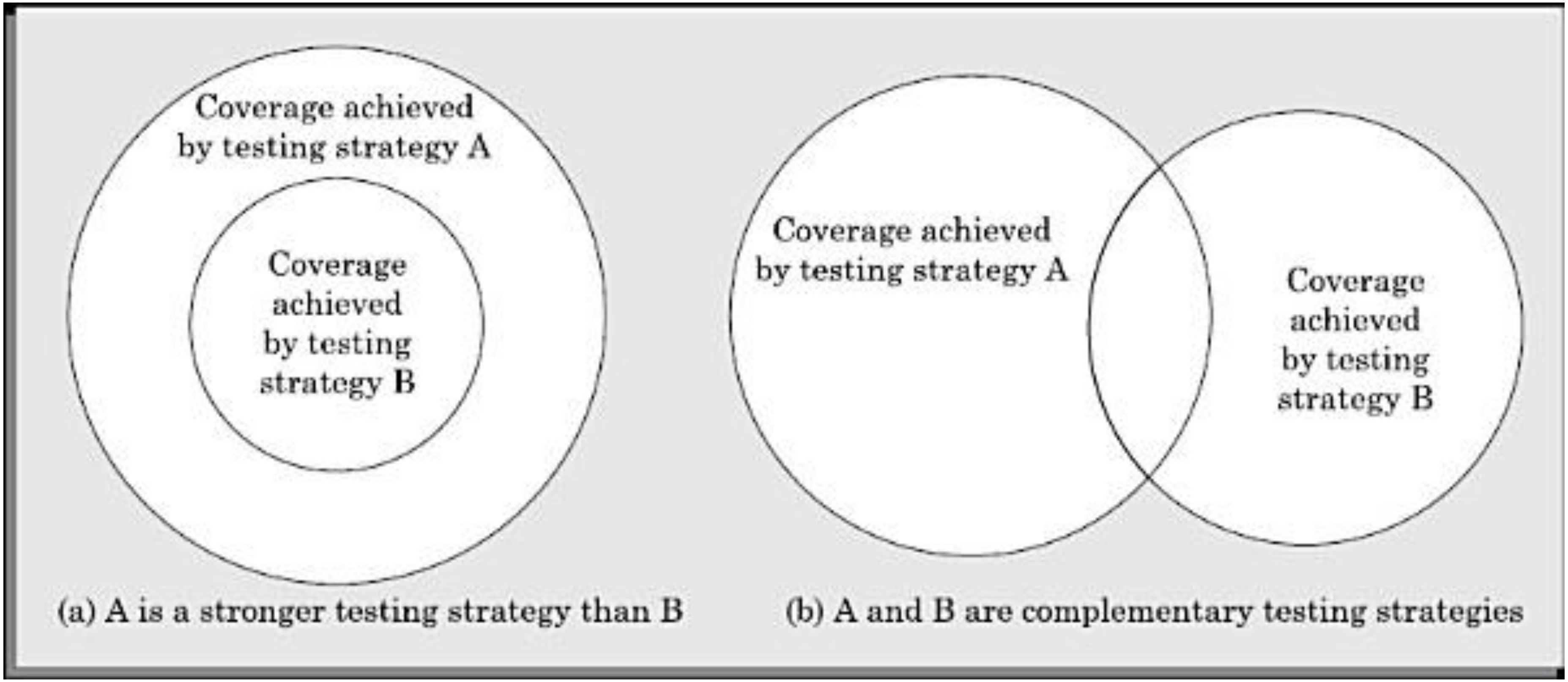
- **Positive test case :** A test case is said to be positive a if it is designed to test whether the software correctly performs a required functionality.
- **Negative test case :** A test case is said to be negative, if it is designed to test whether the software carries out something, that is not required of the system.

- Consider a program to manage user login. A positive test case can be designed to check if a login system validates a user with the correct user name and password. A negative test case in this case can be a test case that checks whether the login functionality validates and admits a user with wrong or bogus login user name or password.

Basic concepts and terminologies

- Test a software using a set of carefully designed test cases:
  - The set of all test cases is called the **test suite.**
- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output.
  - A test case can be said to be an implementation of a test scenario.
- A **test script** is an encoding of a test case as a short program.

- **Testability** The testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.



Basic concepts and terminologies

- A **failure mode** of a software denotes an observable way in which it can fail.
- In other words, all failures that have similar observable symptoms, constitute a failure mode.
  - Example: consider a railway ticket booking software that has three failure modes
    - Failing to book an available seat
    - Incorrect seat booking (e.g., booking an already booked seat)
    - System crash
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode.

Verification versus Validation

- Verification is the process of determining:
  - Whether output of one phase of development conforms to its previous phase.
  - "Are we building the product right”.

- Validation is the process of determining:
  - Whether a fully developed system conforms to its SRS document..
  - "Are we building the right product”.

- Verification is concerned with phase containment of errors:
  - Whereas, the aim of validation is that the final product is error free.

Verification and Validation Techniques

Verification

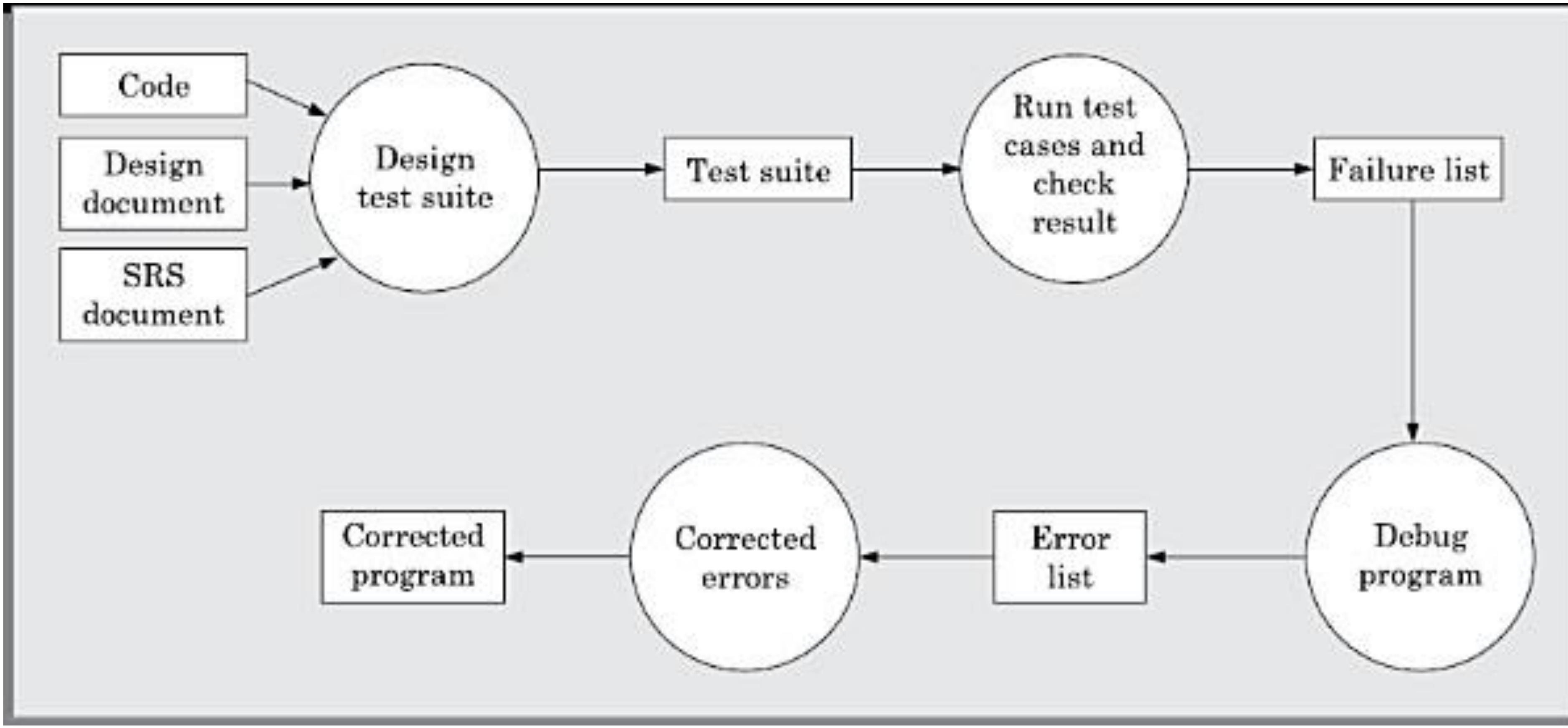
- Review
- Simulation
- Unit testing
- Integration testing

Validation

- System testing

Error detection techniques = Verification techniques + Validation techniques

Testing Activities



Overview of Activities During System and Integration Testing

- Test Suite Design
- Run test cases
- Check results to detect failures.
- Prepare failure list
- Debug to locate errors
- Correct errors.

Tester

Developer

Test Cases

- Each test case typically tries to establish correct working of some functionality:
  - Executes (covers) some program elements.
  - For certain restricted types of faults, fault-based testing can be used.

- **Test case [I,S,O]**
  1. **Set the program in the required state:** Book record created, member record created, Book issued
  2. **Give the defined input:** Select renew book option and request renew for a further 2 week period.
  3. **Observe the output:** Compare it to the expected output.

Sample: Recording of Test Case & Results

- Test Case number
- Test Case author
- Test purpose
- Pre-condition
- Test inputs
- Expected outputs (if any)
- Post-condition
- Test Execution history
  - Test execution date
  - Person executing Test
  - Test execution result (s) : Pass/Fail
    - If failed : Failure information and fix status

Why Design of Test Cases?

- Exhaustive testing of any non-trivial system is impractical:
  - Input data domain is extremely large.
- Design an optimal test suite, meaning:
  - Reasonable size, and uncovers as many errors as possible.
- Systematic approaches are required to design an effective test suite:
  - **Each test case in the suite should target different faults.**

Design of Test Cases

- **The number of test cases in a randomly selected test suite:**
  - **Does not indicate the effectiveness of testing.**

- Consider following example function:

  find-max(int x, int y)

  - Find maximum of two integers x and y.

- The code has a simple programming error:

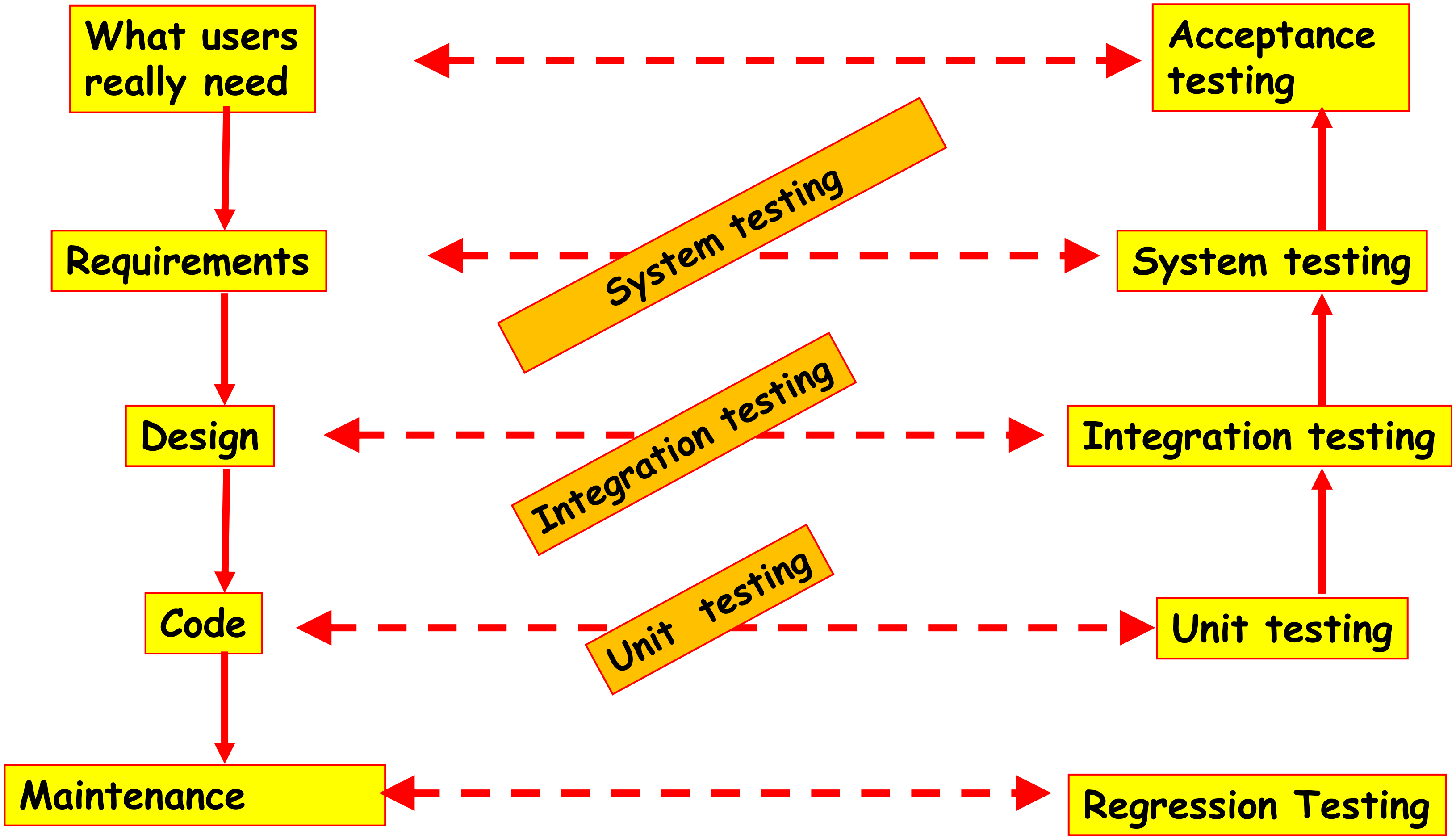
```
If (x>y)
    max = x;
else
    max = x; // should be max=y;
```

- Test suite {(x=3,y=2);(x=2,y=3)} can detect the bug,
- A larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the bug.

4 Testing Levels

- Software tested at 4 levels:
  - Unit testing
    - Test each module (unit, or component) independently
    - **Mostly done by developers of the modules**
  - Integration testing
    - Modules are integrated in steps according to an integration plan
    - The partially integrated system is tested at each integration step.
    - Identifies interface compatibility, unexpected parameter values or state interactions, and run-time exceptions
  - System testing
    - Test the system as a whole
    - **Often done by separate testing or QA team**
  - Regression testing

Levels of Testing



Types of Testing

- Based on types test:
  - **Functionality test**
    - Unit, Integration and system level testing
  - **Performance test**
    - Stress, Volume, Configuration, Compatibility, Regression, Recovery, Maintainance, Documentation, Usability, Security.

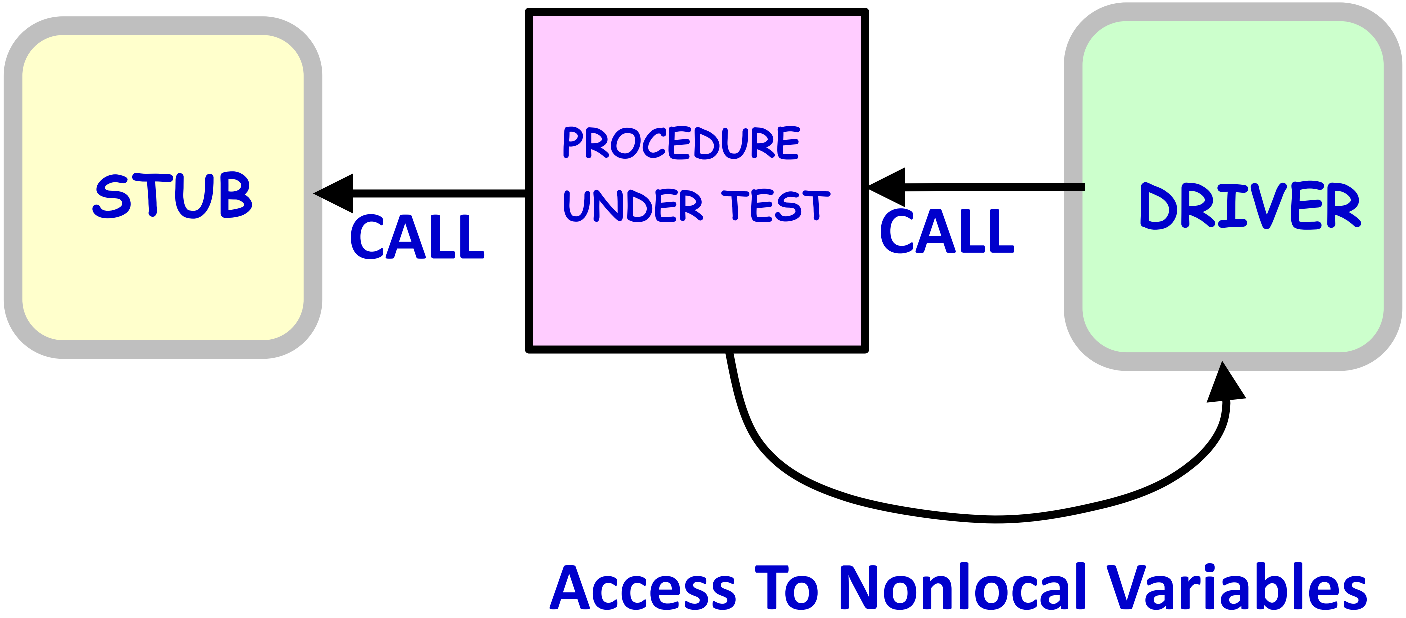
- **Unit Testing**

Unit Testing

- **Testing of individual methods, modules, classes, or components in isolation:**
  - Carried out before integrating with other parts of the software being developed.

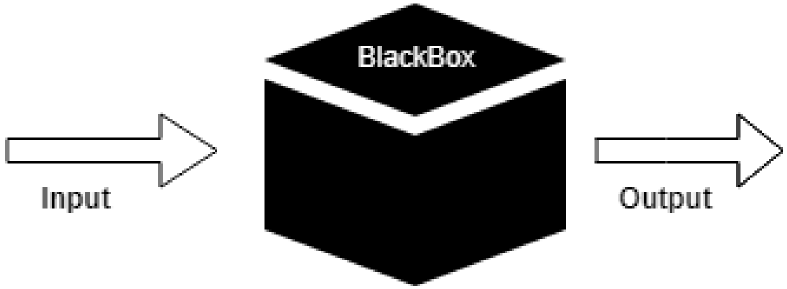
Unit Testing

- Following support required for Unit testing:
  - **Driver:** Simulates the behavior of a function that calls and supplies necessary data to the function being tested.
  - **Stub:** Simulates the behavior of a function that has not yet been written.

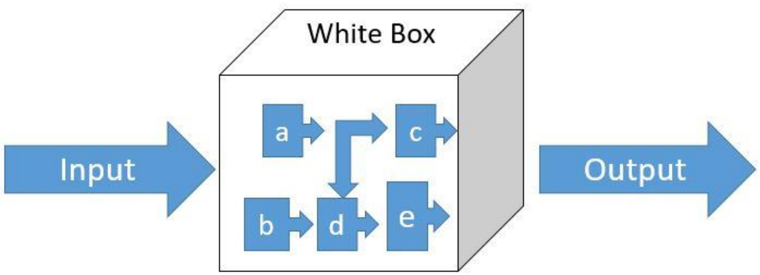


Design of Unit Test Cases

- There are essentially three main approaches to design test cases:
  - **Black-box approach**
    - Equivalence class partitioning
    - Boundary value analysis



  - **White-box (or glass-box) approach**
    - Fault based testing
    - Coverage based testing



  - **Grey-box approach**

- **Black-box testing**

Black Box Testing

- Software considered as a black box:
  - Test data derived from the specification
    - **No knowledge of code necessary**

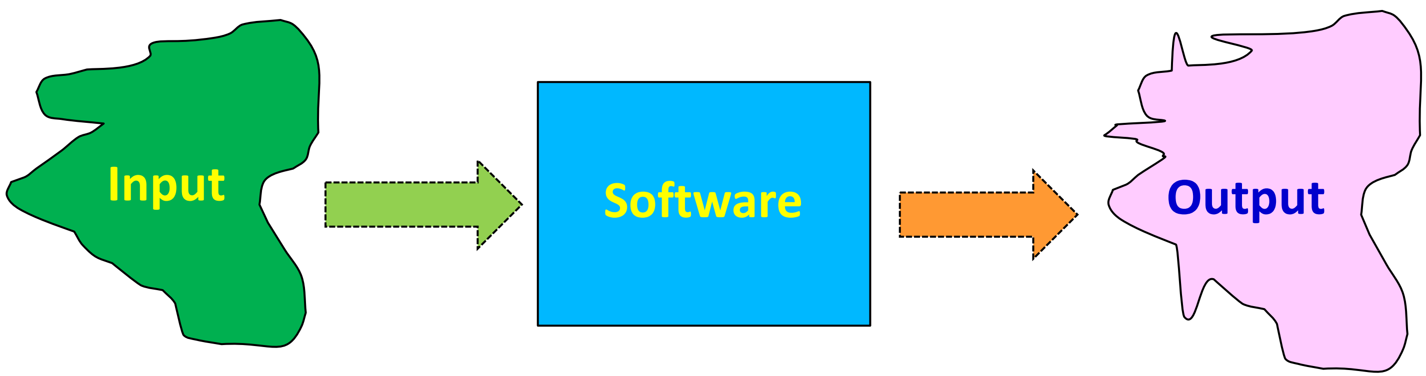
- Also known as:
  - Data-driven or
  - Input/output driven testing

- The goal is to achieve the thoroughness of exhaustive input testing with much less effort!!!!

Black-Box Testing

- Test cases are designed using only **functional specification** of the software:
  - Without any knowledge of the internal structure of the software.

- Black-box testing is also known as **functional testing**.



What is Hard about BB Testing

- Data domain is large

- A function may take multiple parameters:
  - We need to consider the combinations of the values of the different parameters.

- Consider **int check-equal(int x, int y)**
- Assuming a 64 bit computer
  - Input space = 2^{128}
- Assuming it takes 10secs to key-in an integer pair:

  -It would take about a billion years to enter all possible values!

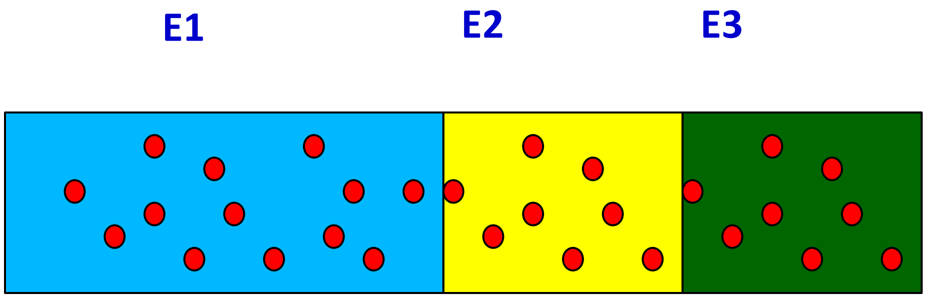
  -Automatic testing has its own problems!

Black Box testing: Equivalence Class Partitioning

- The input values to a program:
  - Partitioned into equivalence classes.
- Partitioning is done such that:
  - **Program behaves in similar ways to every input value belonging to an equivalence class.**
  - **At the least, there should be as many equivalence classes as scenarios.**

Why Define Equivalence Classes?

- Premise:
  - Testing code with any one representative value from a equivalence class:
  - As good as testing using any other values from the equivalence class.

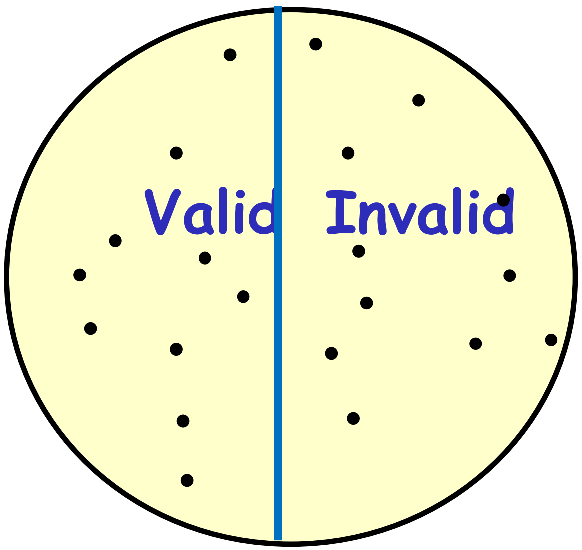


- Example: Given three sides, determine the type of the triangle:
  - Isosceles
  - Scalene
  - Equilateral, etc.

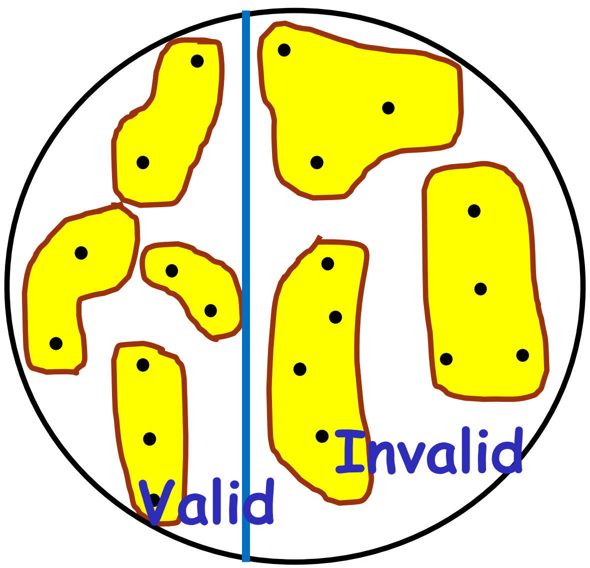
Equivalence Partitioning

- How do you identify equivalence classes?
  - **Identify scenarios**
  - **Examine the input data.**
  - **Examine output**

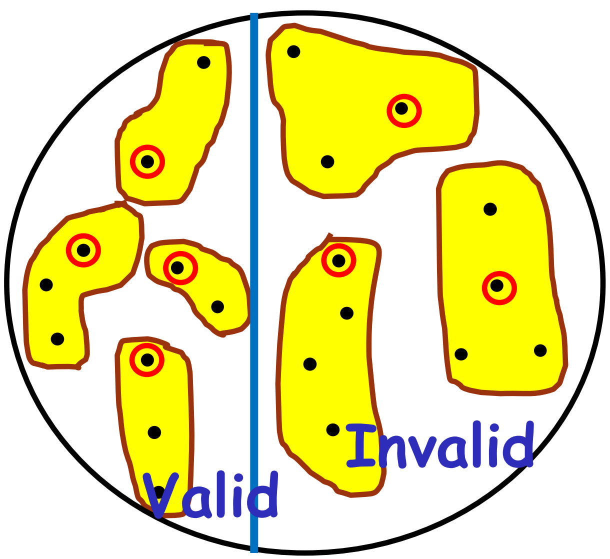
First-level partitioning: Valid vs. Invalid test cases



Further partition valid and invalid test cases into equivalence classes

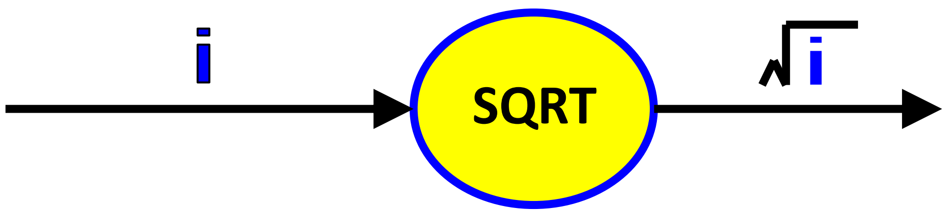


Create a test case for at least one value from each equivalence class



Example (cont.)

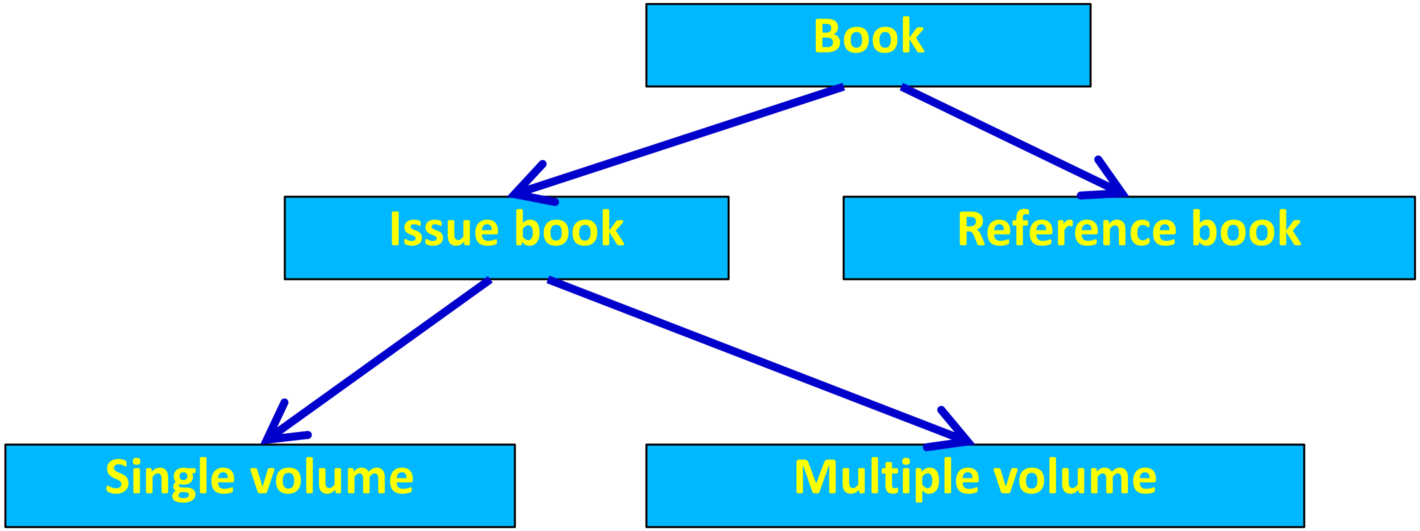
- A program reads an input value in the range of 1 and 5000:
  - Computes the square root of the input number



- Three equivalence classes:
  - The set of negative integers,
  - Set of integers in the range of 1 and 5000,
  - Integers larger than 5000.
  - A possible test suite can be:
    {-5,500,6000}.

Equivalence Partitioning

- **A set of input values constitute an equivalence class if the tester believes these are processed identically:**
  - Example : `issue book(book id);`
  - Different set or sequence of instructions may be executed based on book type.



Black Box testing: Boundary Value Analysis

- Some typical programming errors occur:
  - At boundaries of equivalence classes
  - **Might be purely due to psychological factors.**
- Programmers often fail to see:
  - Special processing required at the boundaries of equivalence classes.
- Boundary value analysis:
  - Select test cases at boundaries of different equivalence classes.

Boundary Value Testing Example

- Process employment applications based on a person's age.

0-16	**Do not hire**
16-18	**May hire on part time basis**
18-55	**May hire full time**
55-99	**Do not hire**

- Notice the problem at the boundaries.
  - Age "16" is included in two different equivalence classes (as are 18 and 55).

Example 1

- For a function that computes the square root of an integer in the range of 1 and 5000:
  - Test cases must include the values: {0,1,2,4999,5000,5001}.

Quiz: BB Test Design

- Design black box test suite for a function that solves a quadratic equation of the form $ax^2+bx+c=0$.
- Equivalence classes
  - Invalid Equation
  - Valid Equation: Roots?

Complex

Real

Coincident

Unique

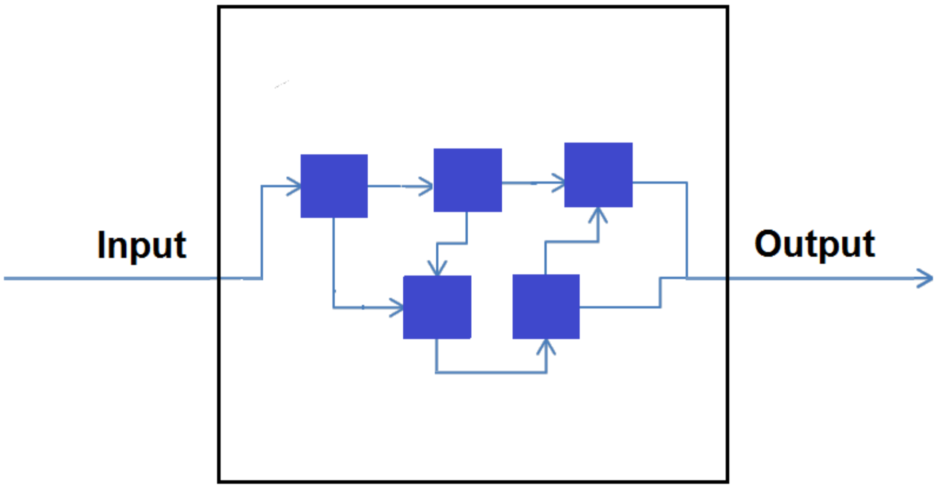
Quiz: Design test Cases

- Customers on a e-commerce site get following discount:
  - A member gets 10% discount for purchases lower than Rs. 2000, else 15% discount
  - Purchase using SBI card fetches 5% discount
  - If the purchase amount after all discounts exceeds Rs. 2000/- then shipping is free.

- **White-Box Testing**

What is White-box Testing?

- White-box test cases designed based on:
  - Code structure of program.
  - White-box testing is also called structural testing.



White-Box Testing Strategies

- **Coverage-based:** Design test cases to cover certain program elements.
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - MCC/MDC coverage
  - Path coverage
  - Data flow-based testing

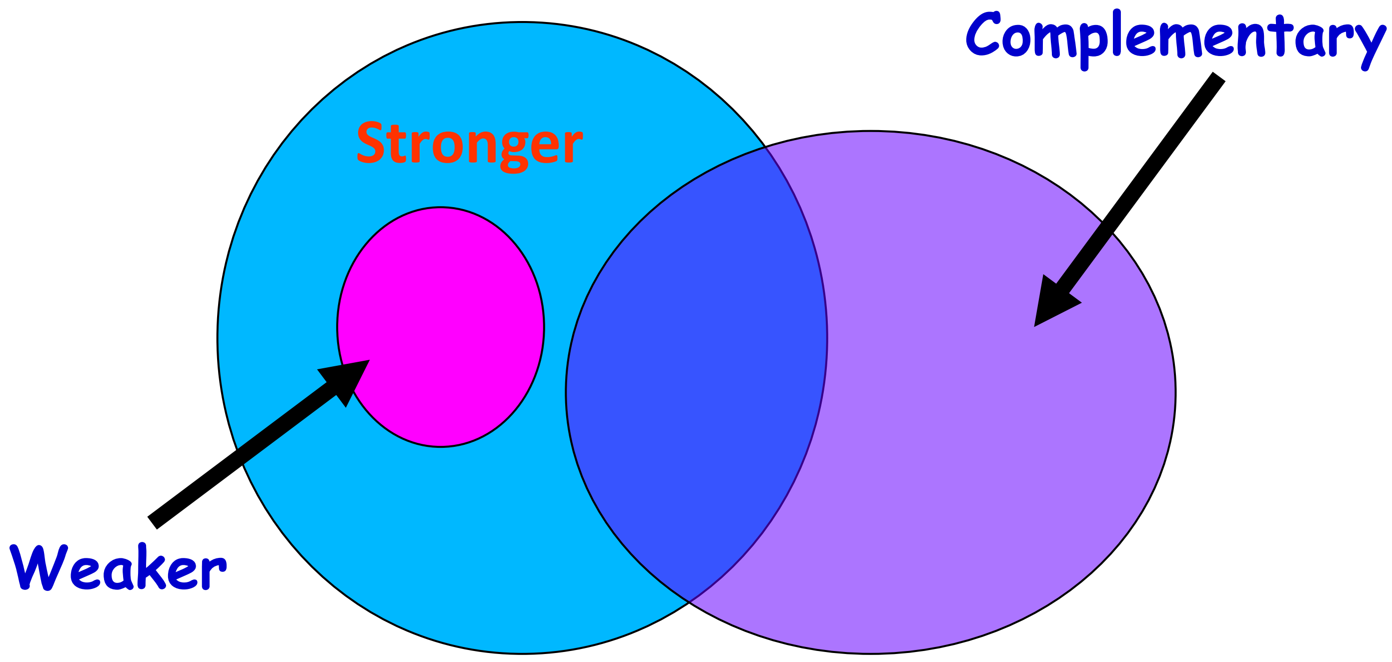
- **Fault-based:** Design test cases to expose some category of faults
  - Mutation testing

Types of program element Coverage

- **Statement:** each statement executed at least once
- **Branch:** each branch traversed (and every entry point taken) at least once
- **Condition:** each condition True at least once and False at least once
- **Multiple Condition:** All combination of Condition covered
- **Path**: each linearly independent path in CFG covered
- **Data flow based:** every definition and uses of data covered

Stronger, Weaker, and Complementary Testing

- Stronger testing: Superset of weaker testing
  - A stronger testing covers all the elements covered by a weaker testing.
  - Covers some additional elements not covered by weaker testing



Statement Coverage

- Statement coverage strategy:
  - Design test cases so that every statement in the program is executed at least once.

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

Statement Coverage

- Coverage measurement:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- **Rationale:** a fault in a statement can only be revealed by executing the faulty statement

- However, observing that a statement behaves properly for one input value:
  - **No guarantee that it will behave correctly for all input values!**

Example

```
int f1(int x, int y){
    while (x != y){
        if (x>y) then
            x=x-y;
        else
            y=y-x;
    }
    return x;
}
```

Euclid's GCD Algorithm

- By choosing the test set

{(x=4,y=3), (x=3,y=4)}

- All statements are executed at least once.

Branch Coverage

- **Also called decision coverage.**
- Test cases are designed such that:
  - Each branch condition
    - Assumes true as well as false value.

- **Adequacy criterion:** Each branch (edge in the CFG) must be executed at least once.

- Coverage:

$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

Example

```
int f1(int x,int y){
    while (x != y){
        if (x>y) then
            x=x-y;
        else
            y=y-x;
    }
  return x;
}
```

Test cases for branch coverage can be:

{(x=3,y=3), (x=4,y=3), (x=3,y=4)}

Branch vs Statement Coverage

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.

- Traversing all edges of a graph causes all nodes to be visited
  - So a test suite that satisfies branch adequacy criterion also satisfies statement adequacy criterion for the same program.

- The converse is not true:
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate).

  - Example: if(x>2)

    x+=1

  x=3 covers the statements but not all branches

All Branches can still miss conditions

- Sample fault: missing operator (negation)

digit_high == 1 && digit_low == -1

- Branch adequacy criterion can be satisfied by varying only digit_high=0, 1
  - **The faulty sub-expression might not be tested!**
  - Even though we test both outcomes of the branch

Condition Coverage

- Basic condition (BCC) coverage .
  - Each basic condition in every conditional expression assumes both true and false values during testing

- Adequacy criterion: each basic condition must be executed at least once.

- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

Basic Condition Coverage

- **Simple or (basic) Condition Testing:**
  - Test cases make each atomic condition to have both T and F values
  - Example: **if (a>10 && b<50)**
  - **The following test inputs would achieve basic condition coverage**
  - **a=15, b=30**
  - **a=5, b=60**
- Does basic condition coverage subsume decision coverage?

Example

- Consider the conditional expression
  - ((A || B) && C):
  - Just two test cases are required
    - {(A=T, B=T, C=F), (A=F, B=F, C=T)}
  - Basic condition coverage may not achieve branch coverage

Multiple condition coverage

- In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values.
- Test cases make Conditions to assume all possible combinations of truth values.

Consider: **if (a || b && c) then …**

Test	a	b	c
1	T	T	T
2	T	T	F
3	T	F	T
4	T	F	F
5	F	T	T
6	F	T	F
7	F	F	T
8	F	F	F

For a composite conditional expression with n components, 2^n test cases are required

MC/DC coverage

- **Modified condition/decision coverage** (MC/DC)

- A test suit would achieve MC/DC if during execution of the test suite each condition in a decision expression independently affects the outcome of the decision.

- Three requirements for MC/DC
  - **Requirement1:** Every decision expression in a program must take both true as well as false values (Same as branch/decision coverage)
  - **Requirement 2:** Every condition in a decision must assume both true and false values (same as BCC)
  - **Requirement 3:** Each condition in a decision should independently affect the decision's outcome

MC/DC Requirement 1

- Every decision expression in a program must take both true as well as false values (Same as branch/decision coverage)

- **The decision is made to take both T/F values.**

```
If ((( a>10 ) && (( b<50 ) && ( c==0 ))) then
```

true

false

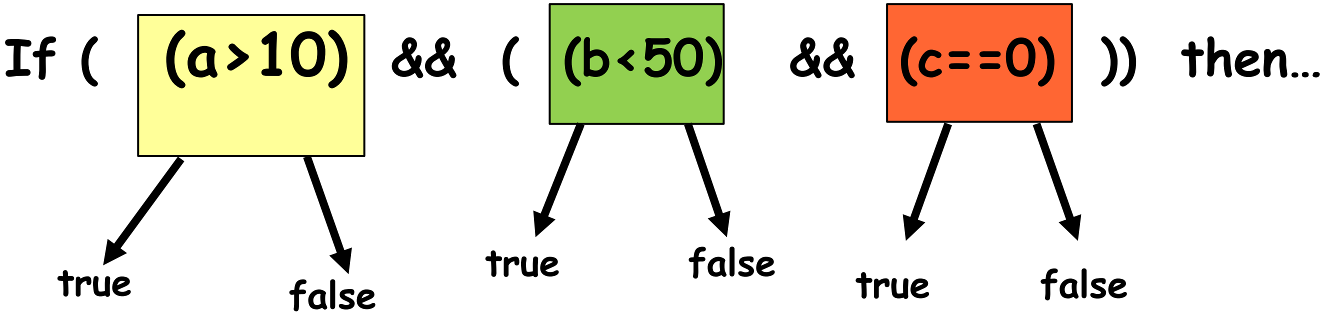
- Test suite: {(a=5, b=10, c=1) , (a=15, b=10, c=0) }

- This is as in Branch coverage.

MC/DC Requirement 2

- Every condition in a decision must assume both true and false values (same as BCC)

- Test cases make every condition in the decision to evaluate to both T and F at least once.



- Test suite: {(a=10, b=10, c=5) , (a=20, b=60, c=0) }

MC/DC Requirement 3

- Each condition in a decision should independently affect the decision's outcome
- Every condition in the decision independently affects the decision's outcome.

If ((a>10) && ((b<50) || (c==0))) then…

true false true false

If ((a>50) && (b<50) || (c==0))) then…

true true false false

If ((a>50) && ((b<50) || (c==0))) then…

true false true false

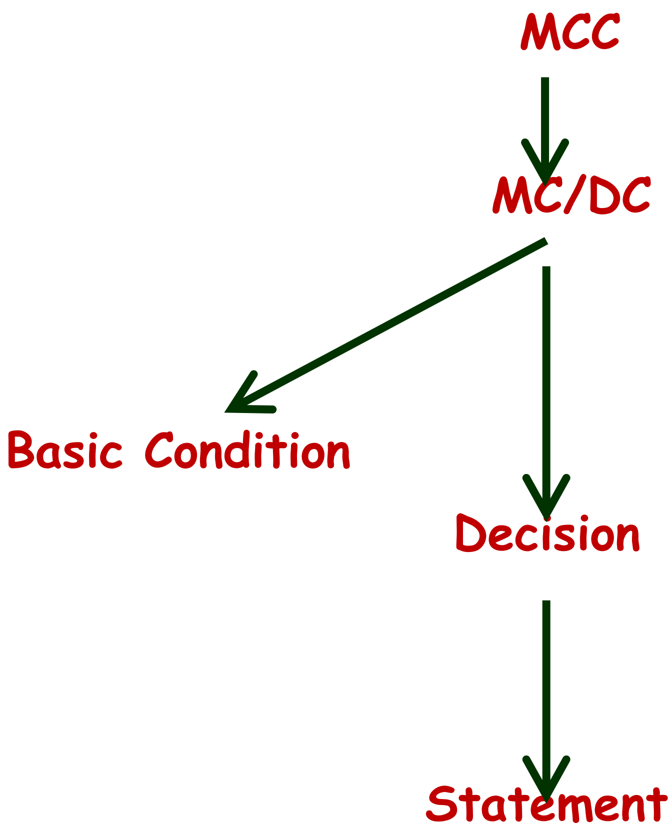
- Test suite: {(a=5, b=30, c=1) , (a=15, b=30, c=1)
  (a=15, b=10, c=1) , (a=15, b=50, c=1)
  (a=60, b=60, c=0) , (a=60, b=60, c=1)
  }

Shortcomings of Condition Testing

- **Redundancy of test cases:** Condition evaluation could be compiler-dependent:
  - Short circuit evaluation of conditions
  - **if(a>30 && b<50)…**
    - If a>30 is FALSE compiler need not evaluate (b<50)
  - Similarly, **if(a>30 || b<50)…**
    - If a>30 is TRUE compiler need not evaluate (b<50)

- **Coverage may be Unachievable:** Possible dependencies among variables:
  - Example: **((chr==`A´)||(chr==`E´))** can not both be true at the same time

Hierarchy



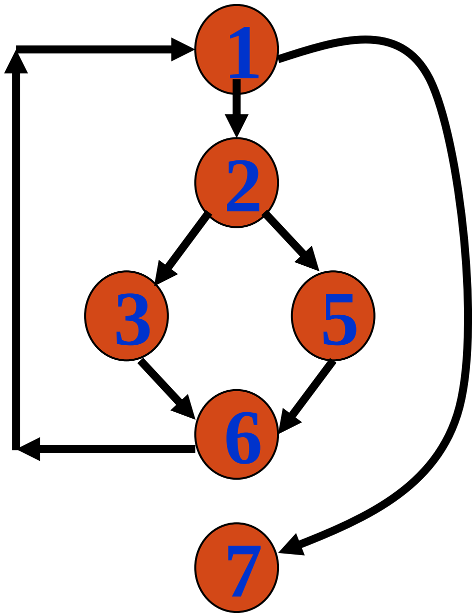
MC/DC: Summary

- MC/DC essentially is :
  - basic condition coverage (C)
  - branch coverage (DC)
  - plus one additional condition (M): every condition must *independently affect* the decision's output
- It is subsumed by MCC and subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- **A good balance of thoroughness and test size and therefore widely used…**

- **Path Testing**

Path Coverage

- Design test cases such that:
  - **All linearly independent paths in the program are executed at least once.**



- Defined in terms of
  - Control flow graph (CFG) of a program.
- A control flow graph (CFG) describes:
  - The sequence in which different instructions of a program get executed.
  - The way control flows through the program.

How to Draw Control Flow Graph?

- **Number all statements of a program.**

- Numbered statements:
  - Represent nodes of control flow graph.

- Draw an edge from one node to another node:
  - **If execution of the statement representing the first node can result in transfer of control to the other node.**

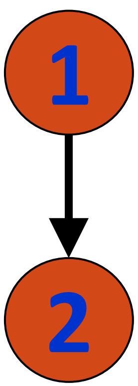
- Every program is composed of:
  - **Sequence**
  - **Selection**
  - **Iteration**

How to Draw Control flow Graph?

- **Sequence:**

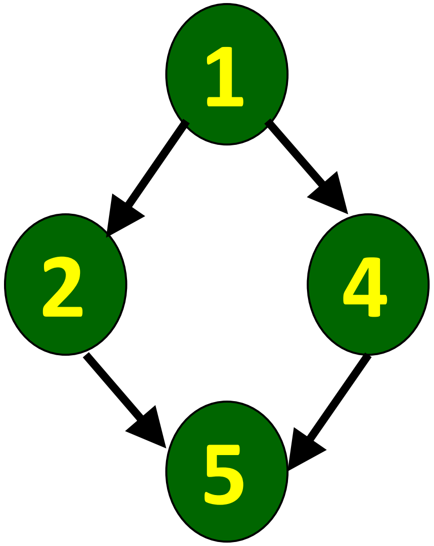
  1 a=5;

  2 b=a*b-1;



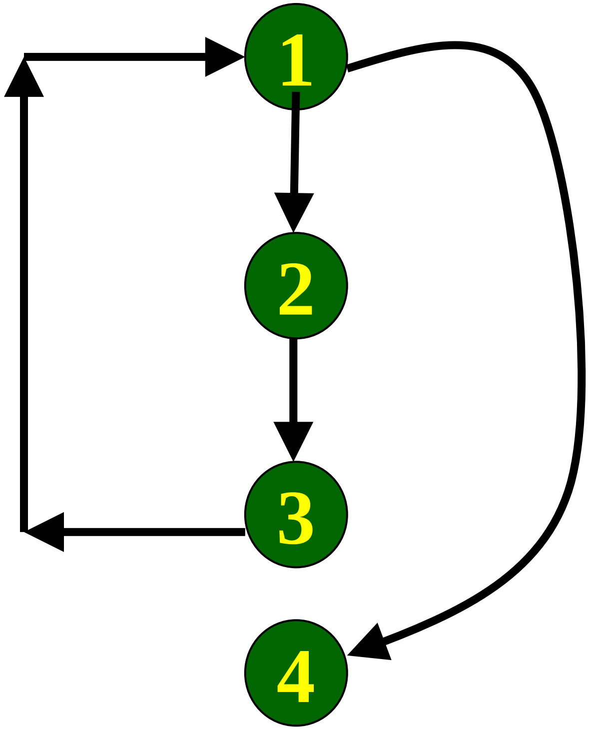
Selection:

1 if(a>b) then
2 c=3;
3 else
4 c=5;
5 c=c*c;



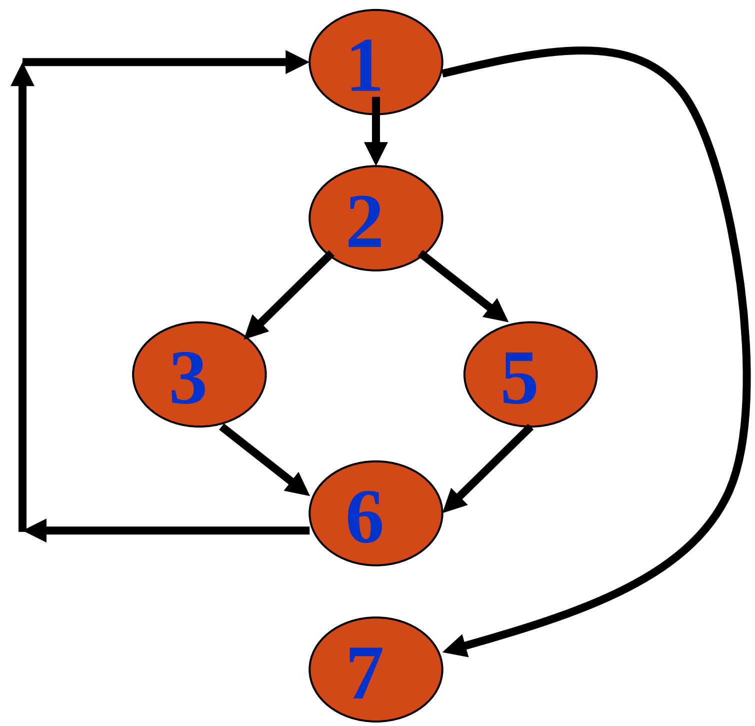
Iteration:

1 while(a>b){
2 b=b*a;
3 b=b-1;}
4 c=b+d;



Example

```
int f1(int x,int y){
1      while (x != y){
2           if (x>y) then
3                x=x-y;
4           else
5                y=y-x;
6      }
7      return x;
}
```



Path

- A path through a program:
  - A node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

- **All path criterion:** In the presence of loops, the number paths can become extremely large:
  - **This makes all path testing impractical**

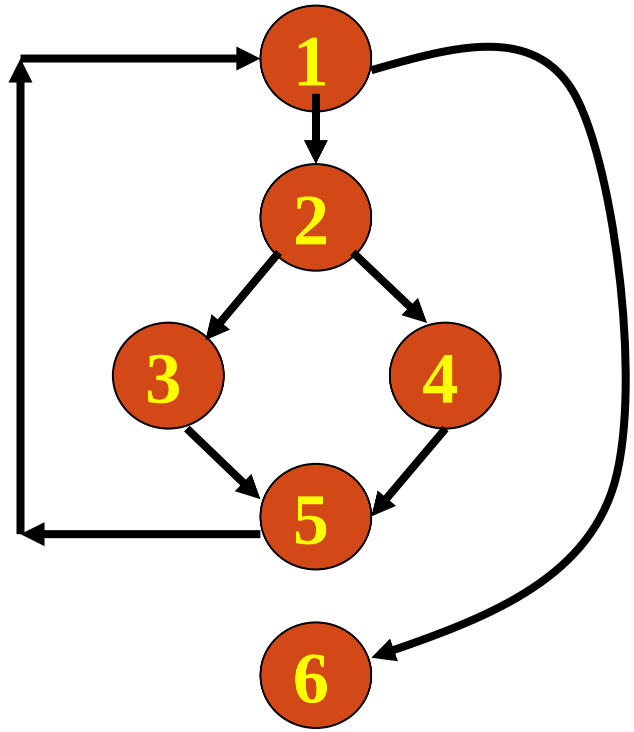
- **Linearly independent path:**
  - Any path through the program that: Introduces at least one new edge:
    - Not included in any other independent paths.

McCabe's Cyclomatic Metric

- For complicated programs:
  - It is not easy to determine the number of independent paths.
- Therefore, McCabe’s cyclomatic Metric is used to estimate upper bound on the linear independent paths
- Provides a practical way of determining:
  - The maximum number of test cases required for basis path testing.
- McCabe's metric provides:
  - **A quantitative measure of testing difficulty and the reliability**

McCabe's Cyclomatic Metric

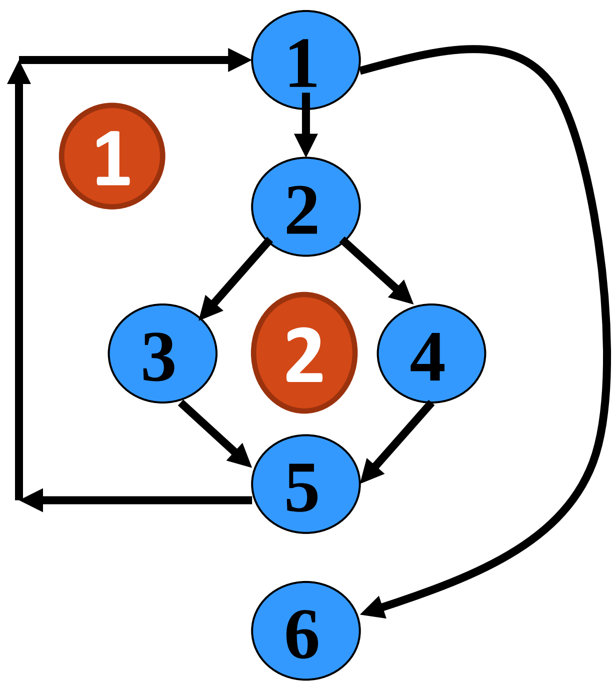
- Given a control flow graph G, cyclomatic complexity V(G):
  - **V(G)= E-N+2**
    - N is the number of nodes in G
    - E is the number of edges in G



Cyclomatic complexity = 7-6+2 = 3.

Cyclomatic Complexity

- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- V(G) = Total number of bounded areas + 1
  - Any region enclosed by a nodes and edge sequence.



From a visual examination of the CFG:

Number of bounded areas is 2.

Cyclomatic complexity = 2+1=3.

Cyclomatic Complexity

- The first method of computing V(G) is amenable to automation:
  - You can write a program which determines the number of nodes and edges of a graph
  - Applies the formula to find V(G).

- Knowing the number of test cases required:
  - Does not make it any easier to derive the test cases,
  - Only gives an indication of the minimum number of test cases required.

- The cyclomatic complexity of a program provides:
  - A upper bound on the number of test cases to be designed
  - To guarantee coverage of all linearly independent paths.

Practical Path Testing

- The tester proposes initial set of test data:
  - Using his experience and judgment.

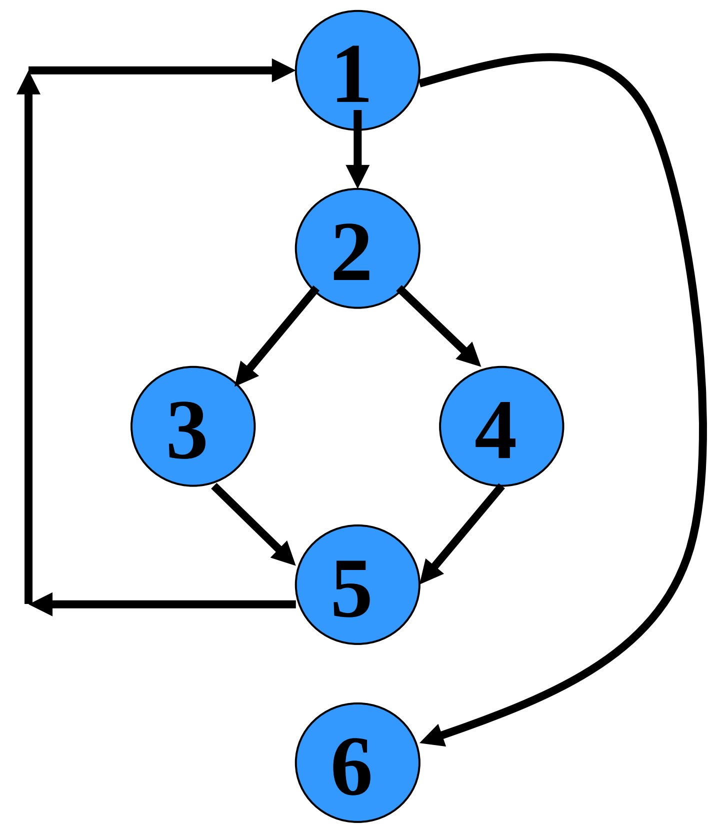
- A dynamic program analyzer used:
  - Measures which parts of the program have been tested
  - Result used to determine when to stop testing.

Derivation of Test Cases

- Draw control flow graph.
- Determine V(G).
- Determine the set of linearly independent paths.
- Prepare test cases:
  - Force execution along each path.
  - Not practical for larger programs.

Example

```
int f1(int x,int y){
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5  }
6 return x;        }
```



Derivation of Test Cases

- Number of independent paths: **3**
  - **1,6** **test case (x=1, y=1)**
  - **1,2,3,5,1,6** **test case(x=1, y=2)**
  - **1,2,4,5,1,6** **test case(x=2, y=1)**

An Interesting Application of Cyclomatic Complexity

- Relationship exists between:
  - McCabe's metric
  - The number of errors existing in the code,
  - The time required to find and correct the errors.

Cyclomatic Complexity

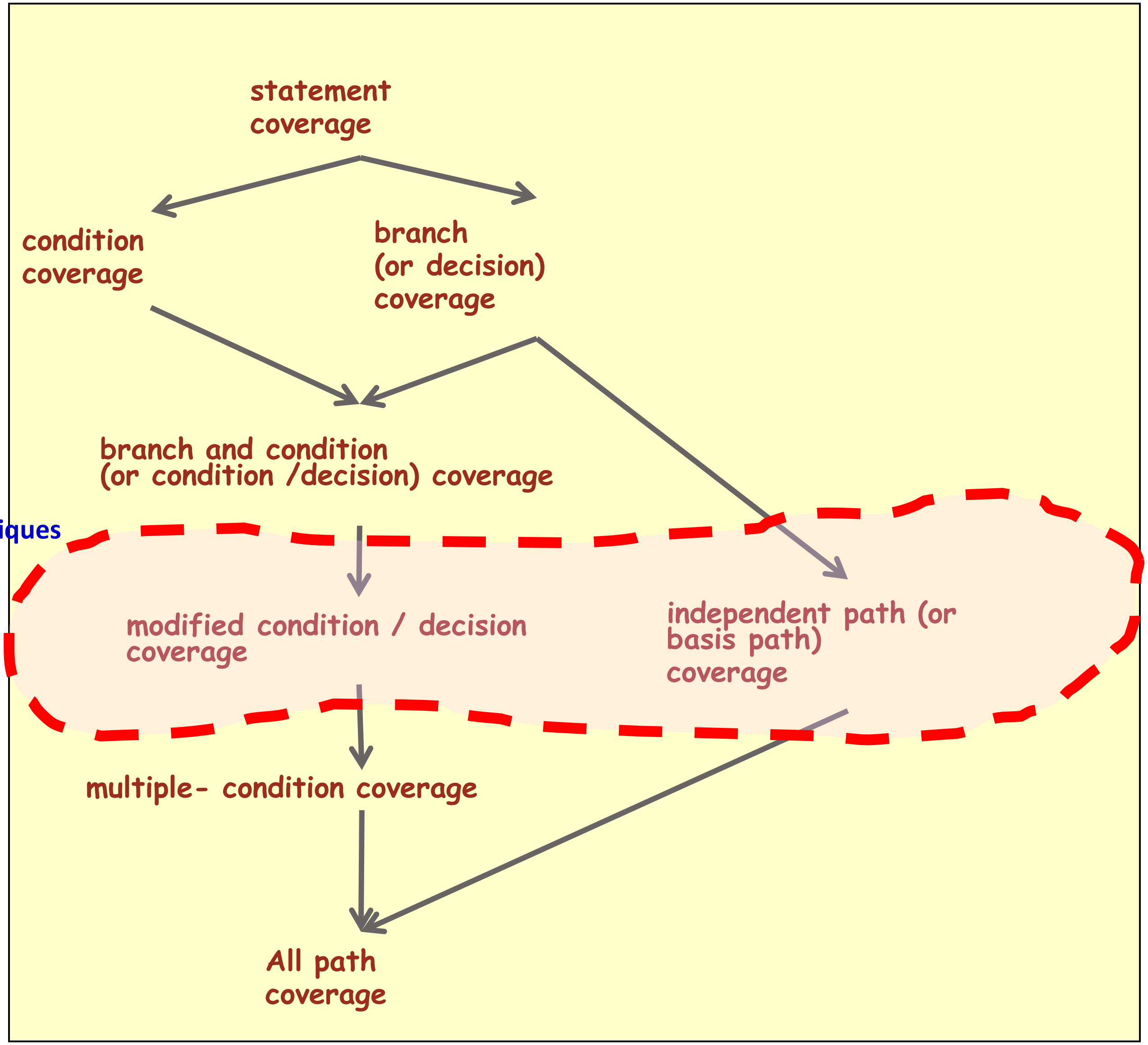
- Cyclomatic complexity of a program:
  - Also indicates the psychological complexity of a program.
  - Difficulty level of understanding the program.
- From maintenance perspective,
  - Limit cyclomatic complexity of modules
    - To some reasonable value.
  - Good software development organizations:
    - Restrict cyclomatic complexity of functions to a maximum of ten or so.

White-Box Testing : Recap

weakest

Practically important coverage techniques

strongest



- **Data flow Testing**

Data Flow-Based Testing

- Select test paths of a program:
  - According to the locations of
    - Definitions and uses of different variables in a program.

```
1 X(){
2     int a=5; /* Defines variable a */
        ....
3  While(c>5) {
4    if (d<50)
5            b=a*a;   /*Uses variable a */
6             a=a-1; /* Defines variable a */
...
7    }
8  print(a); } /*Uses variable a */
```

Data Flow-Based Testing

- For a statement numbered S,
  - DEF(S) = {X/statement S contains a definition of X}
  - USES(S)= {X/statement S contains a use of X}
  - Example: **1: a=b;** DEF(1)={a}, USES(1)={b}.
  - Example: 2: **a=a+b;** DEF(1)={a}, USES(1)={a,b}.

Definition-use chain (DU chain)

- [X,S,S1],
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - the definition of X in the statement S is live at statement S1.

There exist a path from S to S1 not containing any definition of X

DU Chain Example

A variable X is said to be live at statement S1, if

X is defined at a statement S:

There exists a path from S to S1 not containing any definition of X.

```
1 X(){
2  int a=5; /* Defines variable a */
3  While(c>5) {
4     if (d<50)
5         b=a*a;   /*Uses variable a */
6        c=a+5;
7         a=a-1; /* Defines variable a */
8    }
9  print(a); } /*Uses variable a */
```

[a,2,5]
[a,2,6]
[a,7,7]

Data Flow-Based Testing

- One simple data flow testing strategy:
  - **Every DU chain in a program be covered at least once.**
- Data flow testing strategies:
  - Useful for selecting test paths of a program containing nested if and loop statements.

Data Flow-Based Testing

```
X(){
 B1;      /* Defines variable a */
 While(C1) {
    if (C2)
          if(C4) B4; /*Uses variable a */
       else B5;
          else  if (C3) B2;
       else B3;    }
 B6 }
```

Data Flow-Based Testing

- [a,2,5]: a DU chain.
- Assume:
  - DEF(X) = {B1, B2, B3, B4, B5}
  - USES(X) = {B2, B3, B4, B5, B6}
  - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

- **Mutation Testing**

Mutation Testing

- In this, software is first tested:
  - Using an initial test suite designed using white-box strategies we already discussed.

- After the initial testing is complete,
  - Mutation testing is taken up.

- The idea behind mutation testing:
  - **Make a few arbitrary small changes to a program at a time.**

Main Idea

- Insert faults into a program:
  - Check whether the test suite is able to detect these.
  - This either validates or invalidates the test suite.

- Each time the program is changed:
  - It is called a **mutated program**
  - The change is called a **mutant.**

Mutation Testing

- A mutated program:
  - Tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - A mutant gives an incorrect result,
  - **Then the mutant is said to be dead.**
- If a mutant remains alive:
  - Even after all test cases have been exhausted,
  - **The test suite is enhanced to kill the mutant.**
- The process of generation and killing of mutants:
  - **Can be automated by predefining a set of primitive changes that can be applied to the program.**

Mutation Testing

- Example primitive changes to a program:
  - Deleting a statement
  - Altering an arithmetic operator,
  - Changing the value of a constant,
  - Changing a data type, etc.

Traditional Mutation Operators

- Deletion of a statement
- Boolean:
  - Replacement of a statement with another

    eg. **==** and **>=, <** and **<=**
  - Replacement of boolean expressions with *true* or *false* eg. **a || b** with ***true***
- Replacement of arithmetic

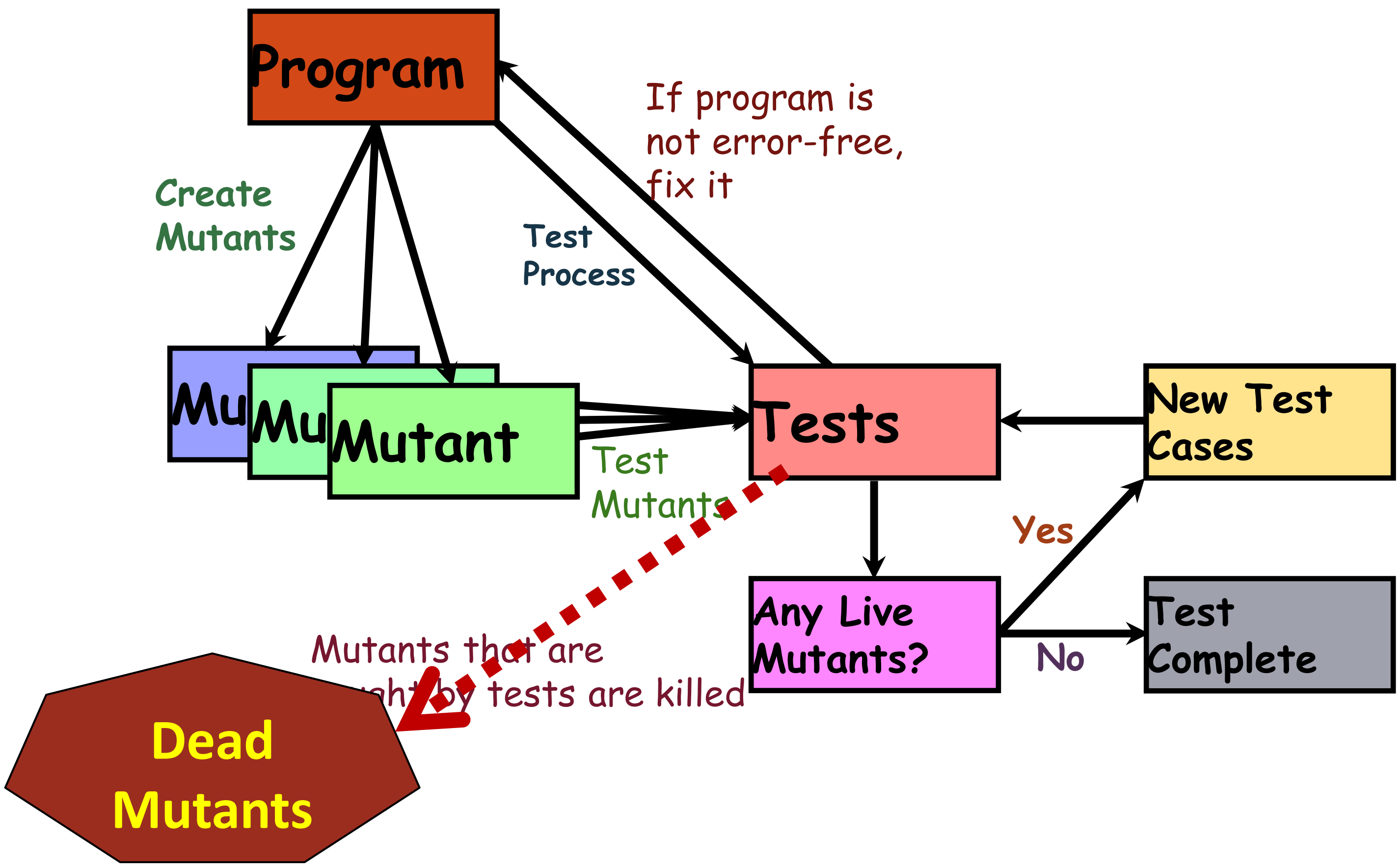
  eg. * and +, / and -
- Replacement of a variable (ensuring same scope/type)

Underlying Hypotheses

- Mutation testing is based on the following two hypotheses:
  - **The Competent Programmer Hypothesis**
    - Programmers create programs that are close to being correct:
      - Differ from the correct program by some simple errors.
  - **The Coupling Effect**

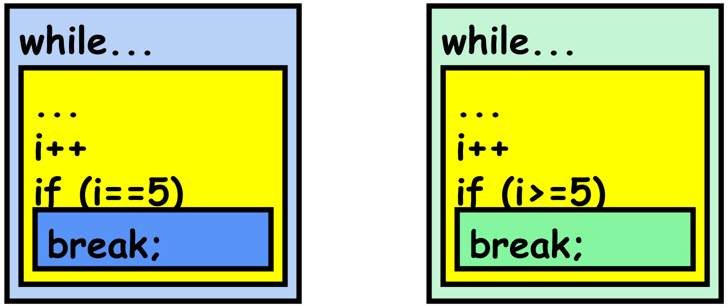
    Both of these were proposed by DeMillo *et al.*,1978
    - **Complex errors are caused due to several simple errors.**
    - It therefore suffices to check for the presence of the simple errors

The Mutation Process



Equivalent Mutants

- There may be surviving mutants that cannot be killed,
  - These are called Equivalent Mutants
- Although syntactically different:
  - These mutants are indistinguishable through testing.
- Therefore have to be checked 'by hand'



Disadvantages of Mutation Testing

- Equivalent mutants
- Computationally very expensive.
  - A large number of possible mutants can be generated.
- Certain types of faults are very difficult to inject.
  - Only simple syntactic faults introduced

Quiz 1

- Identify one advantage and one disadvantage of the mutation test technique.

- Identify two advantages and two disadvantages of the mutation test technique.
- **Adv**:
  - Can be automated
  - Helps effectively strengthen black box and coverage-based test suite
- **Disadv:**
  - Equivalent mutants

Why Both BB and WB Testing?

Black-box .

- Impossible to write a test case for every possible set of inputs and outputs
- Some code parts may not be reachable
- Does not tell if extra functionality has been implemented.

White-box .

- Does not address the question of whether a program matches the specification
- Does not tell if all functionalities have been implemented
- Does not uncover any missing program logic

- **Debugging**

Debugging

- Once errors are identified:
  - **Debug:** Identify precise location of errors
  - **Fix errors**
  - **Regression test**
- Each debugging approach has its advantages and disadvantages:
  - Each is useful in appropriate circumstances.

Testing vs Debugging

TESTING	DEBUGGING
Activity to check whether the actual results match the expected results of the software and to ensure that it is defect-free	Process of finding and resolving defects or problems within a computer program, which prevent correct operation of computer software or a system
Process of finding and locating defects of the software	Process of fixing the identified defects
Performed by the testing team	Performed by the development team
Purpose is to find many defects as possible	Purpose is to remove the detected defects

Ref: https://pediaa.com/difference-between-testing-and-debugging/

Debugging approaches

- Brute force methods
- Symbolic debugging
- Backtracking
- Cause elimination method
- Program slicing

Brute-Force Method

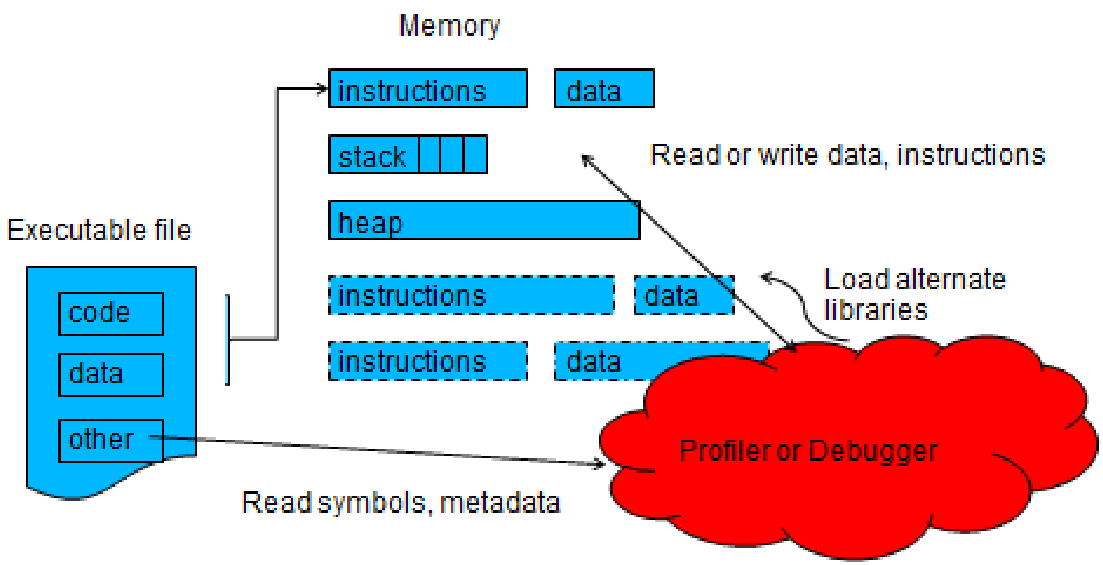
- This is the most common method of debugging:
  - Least efficient method
  - Program is loaded with print statements
  - Print the intermediate values
  - Hope that some of printed values will help identify the error

Symbolic Debugger

- More systematic than the Brute force method:
  - Symbolic debuggers get their name for historical reasons
  - Early debuggers supported only examination of values from a program dump:
    - Tedious to determine which variable a byte sequence corresponds to.

Symbolic Debugger

- Symbolic debugging involves directly inspecting the state of a running program, using debugging symbols embedded in the executable to correlate memory locations or stack frames to specific variables or lines of code.
- Likewise, symbolic debuggers are capable of controlling the execution of an application; stopping it at certain points for inspection, or slowly stepping through its execution instruction-by-instruction so that its control flow can be observed.



Ref: https://cvw.cac.cornell.edu/Profiling/debugging_runtime_symbolic

Backtracking

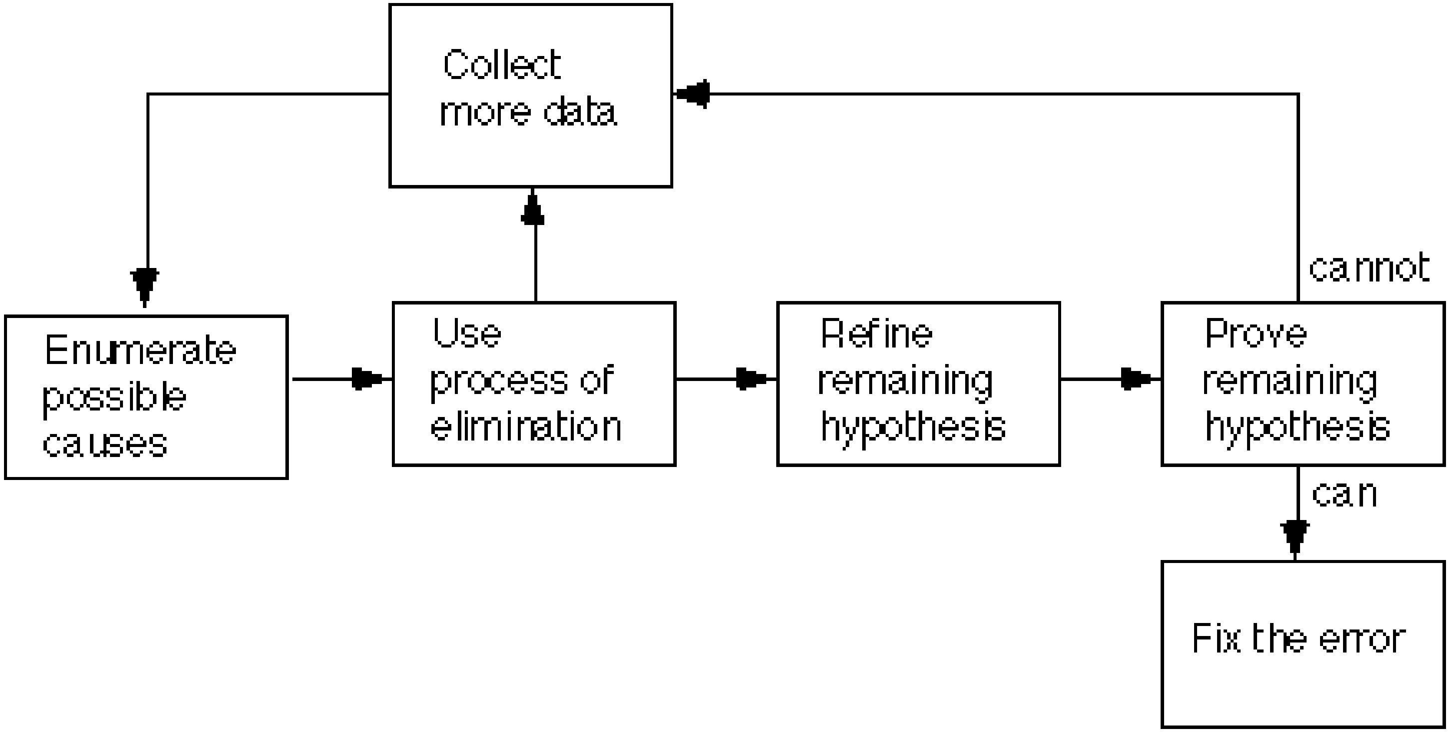
- This is a fairly common approach.
- Beginning at the statement where an error symptom has been observed:
  - Source code is traced backwards until the error is discovered.

```
int main(){
int i,j,s;
i=1;
while(i<=10){
        s=s+i;
        i++; j=j++;}
printf("%d",s);
}
```

Cause elimination method

- Once a failure is observed, the symptoms of the failure are noted.

- Based on the symptoms, the causes which could possibly have contribute to the symptom is identified and test area conducted to eliminate each



Img ref: https://jodypaul.com/SWE/TD/TestDebug.html

Program slicing

- Similar to backtracking with a modification:
  - The search space is reduced by defining slice
  - A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

1 `scanf("%d",&n);`	1 `scanf("%d",&n);`
2 `sum=0;`	2 `sum=0;`
3 `product=1;`	3
4 `while (n>0)`	4 `while (n>0)`
5 `{`	5 `{`
6 `sum=sum+n;`	6 `sum=sum+n;`
7 `product=product*n;`	7
8 `n=n-1;`	8 `n=n-1;`
9 `}`	9 `}`
p: Original Program	p': Slice of p w.r.t. *sum* at line 9

Img ref: Mark Harman, David Binkley, Sebastian Danicic, Amorphous program slicing, Journal of Systems and Software, Volume 68, Issue 1, Pages 45-64, 2003,

Debugging Guidelines

- Debugging requires a thorough understanding of program design.
- Debugging may sometimes require full redesign of the program.
- A common mistake novice programmers often make:
  - Not fixing the error but the error symptoms.

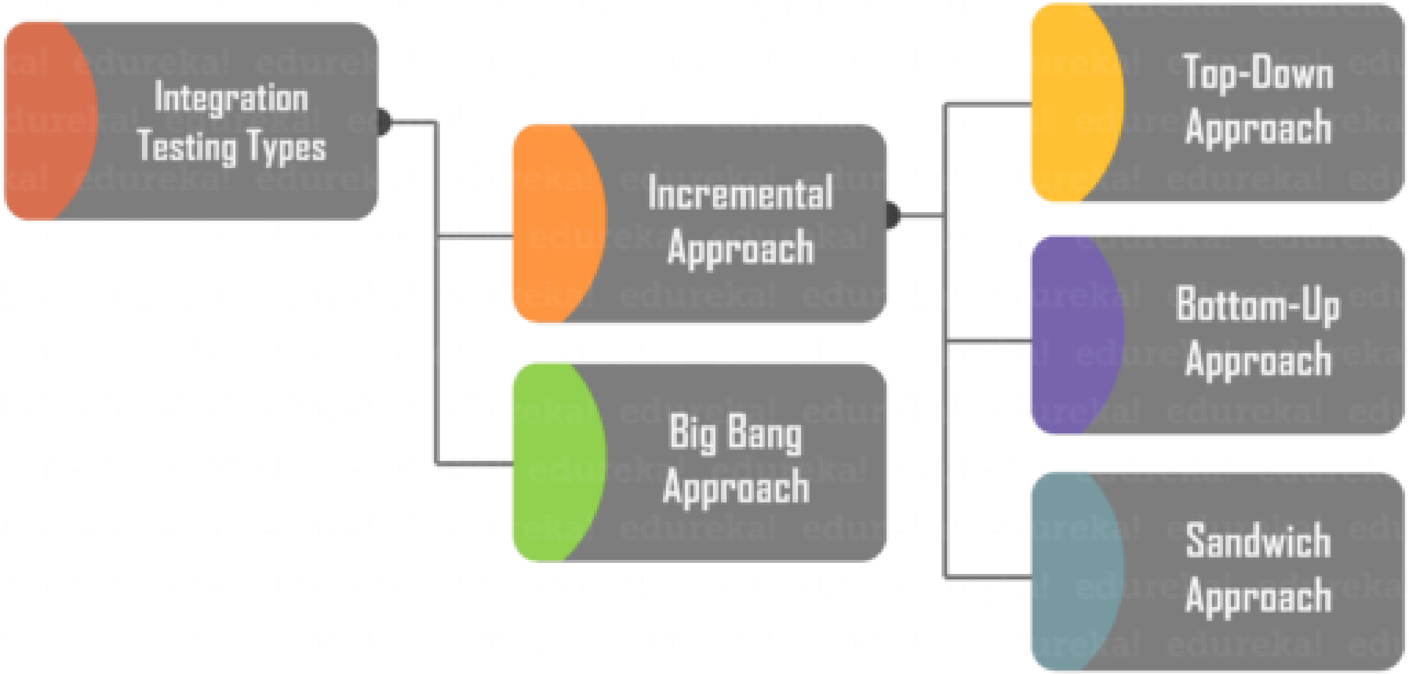
Program Analysis Tools

- There are essentially two categories of program analysis tools:
  - **Static analysis tools**
    - Assess properties of a program without executing it.
    - Analyze the source code to certain analytical conclusions.
    - Check whether: Coding standards been adhered to? Commenting is adequate?
    - Programming errors such as:
      - Uninitialized variables
      - Mismatch between actual and formal parameters.
      - Variables declared but never used, etc.
  - **Dynamic analysis tools**
    - Require the program to be executed:
    - Its behavior recorded.
    - Produce reports such as adequacy of test cases.

- **Integration testing**

Integration Testing Approaches

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach



Big Bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - All the modules are simply put together and tested.
  - This technique is used only for very small systems.

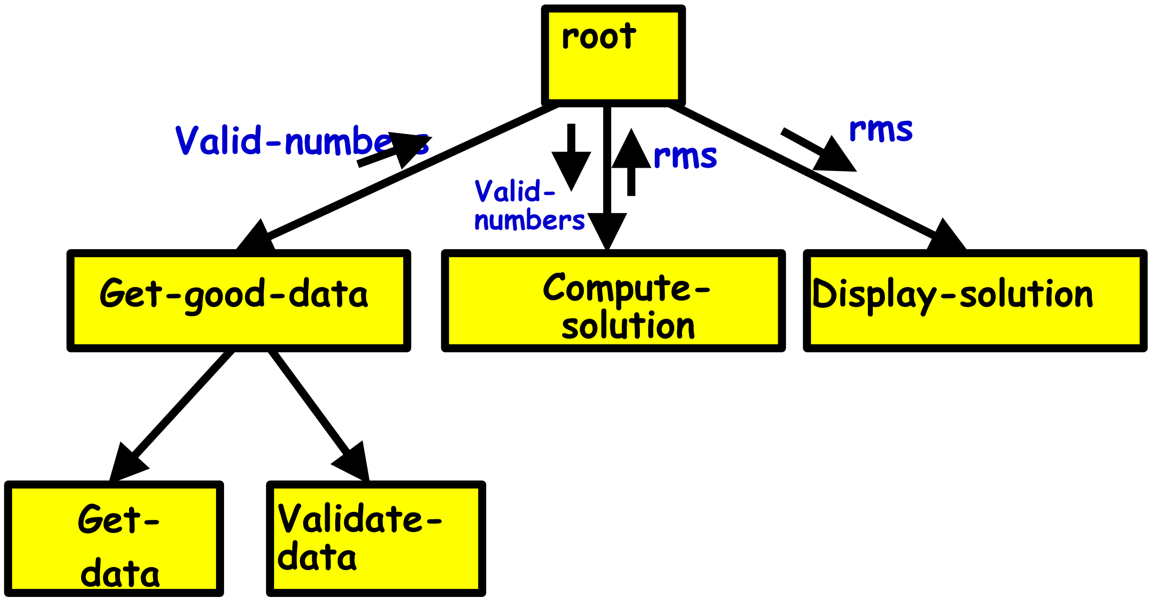
- Main problems with this approach:
  - If an error is found:
    - It is very difficult to localize the error
    - The error may potentially belong to any of the modules being integrated.
  - Debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

- Integrate and test the bottom level modules first.

- A disadvantage of bottom-up testing:
  - When the system is made up of a large number of small subsystems.
  - This extreme case corresponds to the big bang approach.

  - Testing can start only after bottom level modules are ready.



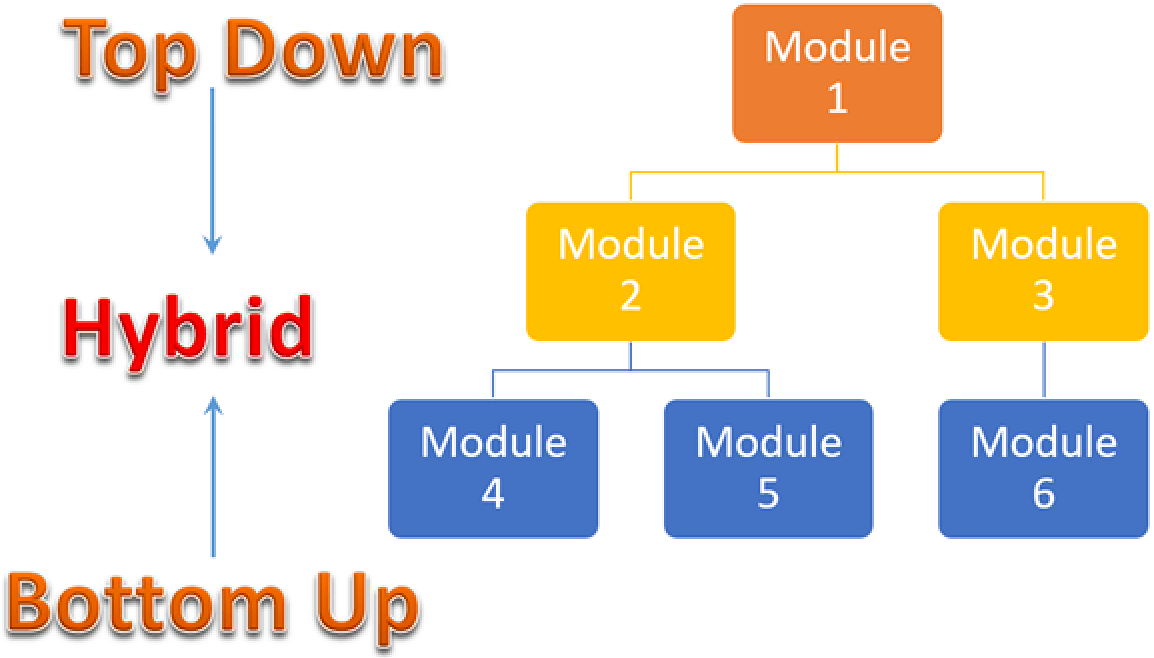
Top-down Integration Testing

- Top-down integration testing starts with the main routine and one or two subordinate routines in the system.

- After the top-level 'skeleton’ has been tested:
  - immediate subordinate modules of the 'skeleton’ are combined with it and tested.
  - Testing waits till all top-level modules are coded and unit tested.

Mixed Integration Testing

- Mixed (or sandwiched) integration testing:
  - Uses both top-down and bottom-up testing approaches.
  - **Most common approach**



- **System testing**

System Testing

- Objective:
  - **Validate a fully developed software against its requirements.**

- There are three main types of system testing:
  - **Alpha Testing:** System testing carried out by the test team within the developing organization.
    - Test cases are designed based on the SRS document
  - **Beta Testing:** System testing performed by a select group of friendly customers.
  - **Acceptance Testing:** System testing performed by the customer himself:
    - **To determine whether the system should be accepted or rejected.**

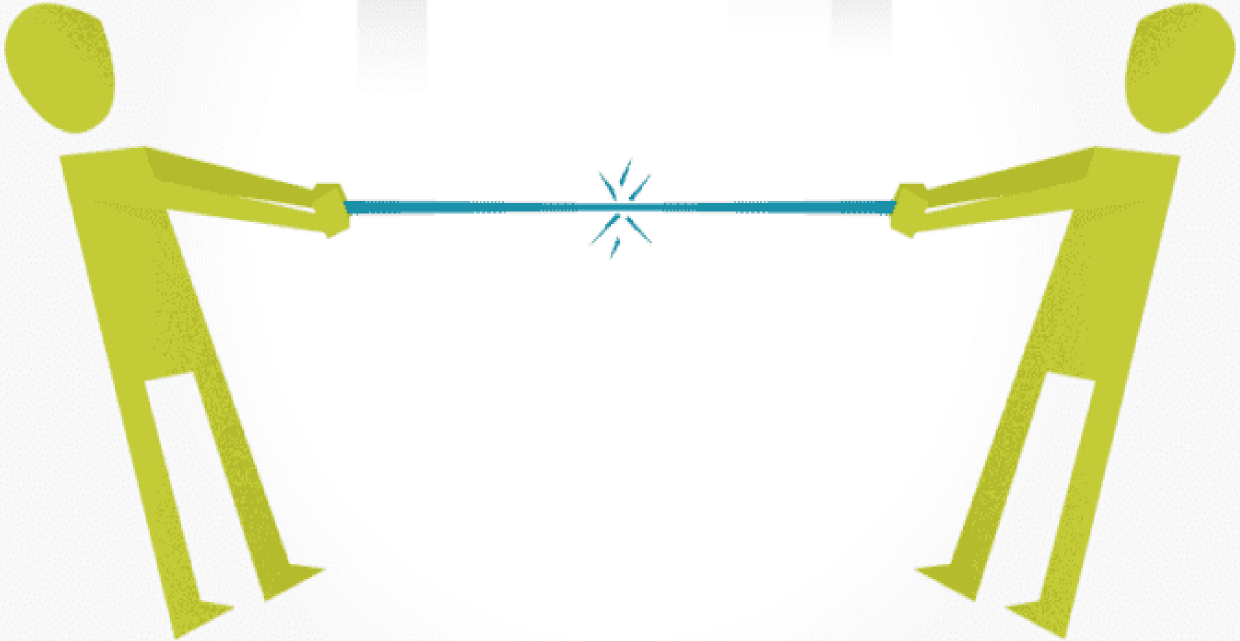
Performance Testing

- Addresses non-functional requirements.
  - May sometimes involve testing hardware and software together.
  - There are several categories of performance testing.

- Performance testing
  - Stress testing
  - Volume testing
  - Configuration testing
  - Compatibility testing
  - Recovery testing
  - Maintenance testing
  - Documentation testing

Stress Testing

- Stress testing (also called endurance testing):
  - Impose abnormal input to stress the capabilities of the software.
  - **Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity.**



Stress Testing

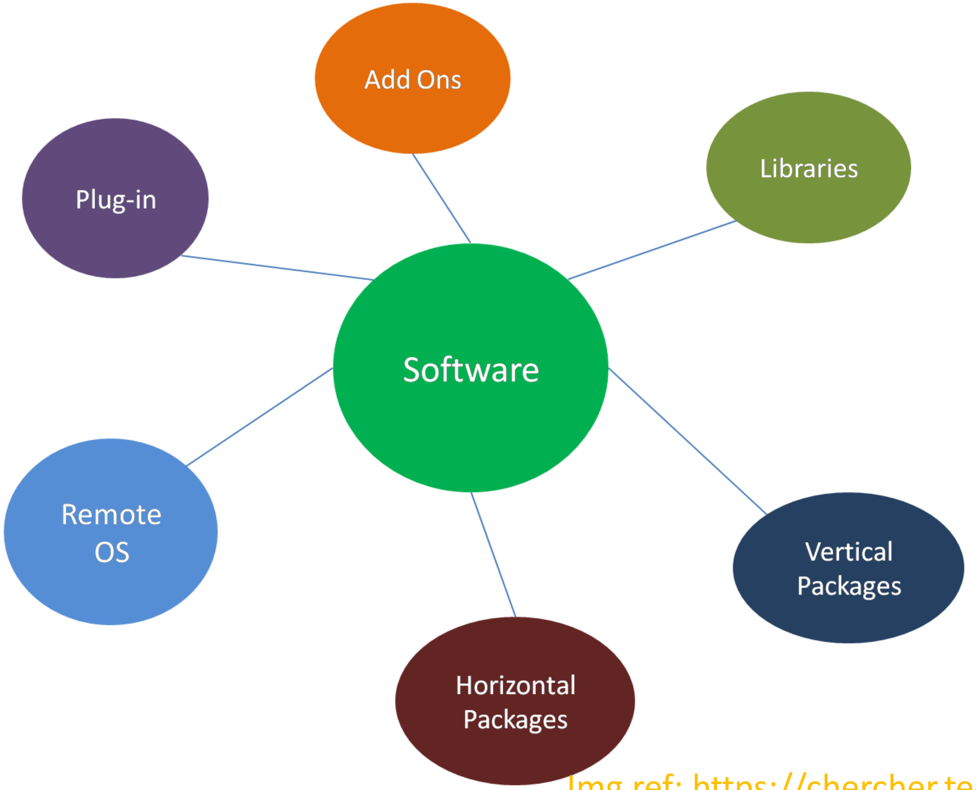
- If the requirements is to handle a specified number of users, or devices:
  - Stress testing evaluates system performance when all users or devices are busy simultaneously.
- If an operating system is supposed to support 15 multiprogrammed jobs,
  - The system is stressed by attempting to run 15 or more jobs simultaneously.
- A real-time system might be tested
  - To determine the effect of simultaneous arrival of several high-priority interrupts.
- Stress testing usually involves an element of time or size,
  - Such as the number of records transferred per unit time,
  - The maximum number of users active at any time, input data size, etc.
- Therefore stress testing may not be applicable to many types of systems.

Volume Testing

- Addresses handling large amounts of data in the system:
  - Whether data structures (e.g. queues, stacks, arrays, etc.) are large enough to handle all possible situations.
  - Fields, records, and files are stressed to check if their size can accommodate all possible data volumes.

Configuration Testing

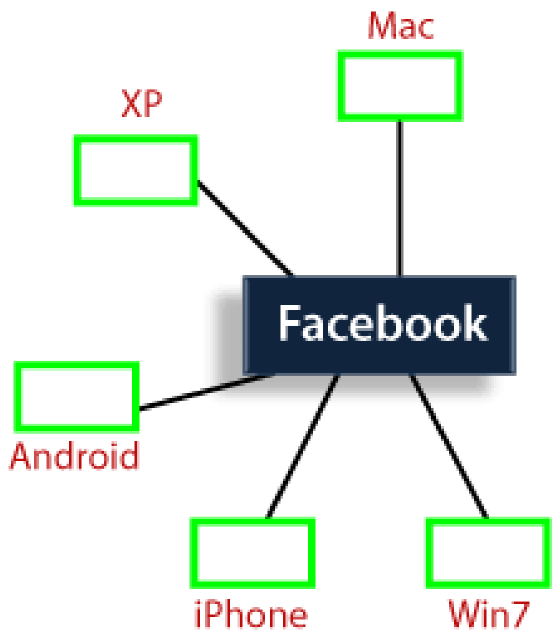
- Analyze system behavior:
  - in various hardware and software configurations specified in the requirements
  - sometimes systems are built in various configurations for different users
  - for instance, a minimal system may serve a single user,
    - other configurations for additional users.



Img ref: https://chercher.tech/testing/configuration-testing

Compatibility Testing

- These tests are needed when the system interfaces with other systems:
  - Check whether the interface functions as required.



- If a system is to communicate with a large database system to retrieve information:
  - A compatibility test examines speed and accuracy of retrieval.

Recovery Testing

- These tests check response to:
  - Presence of faults or to the loss of data, power, devices, or services
  - Subject system to loss of resources
    - Check if the system recovers properly.

Maintenance Testing

- Diagnostic tools and procedures:
  - help find source of problems.
  - It may be required to supply
    - memory maps
    - diagnostic programs
    - traces of transactions,
    - circuit diagrams, etc.

- Verify that:
  - all required artifacts for maintenance exist, they function properly

Documentation tests

- Check that required documents exist and are consistent:
  - user guides,
  - maintenance guides,
  - technical documents

- Sometimes requirements specify:
  - Format and audience of specific documents
  - Documents are evaluated for compliance

Error seeding

- Error seeding technique is used to estimate the number of residual errors in a software

- Make a few arbitrary changes to the program:
  - Artificial errors are seeded into the program.
  - Check how many of the seeded errors are detected during testing.

- The kinds of seeded errors should match closely with existing errors:
  - However, it is difficult to predict the types of errors that exist.
- Categories of remaining errors:
  - **Can be estimated by analyzing historical data from similar projects.**

Error seeding

- Let:
  - N be the total number of errors in the system
  - n of these errors be found by testing.
  - S be the total number of seeded errors,
  - s of the seeded errors be found during testing.

- n/N = s/S
- N = S x n/s
- remaining defects:

  N - n = n x ((S - 1)/ s)

EXAMPLE:

- 100 errors were introduced.
- 90 of these errors were found during testing
- 50 other errors were also found.
- Remaining errors=
  50 (100-90)/90 = 6

Quiz 3

- Before system testing 100 errors were seeded.
- During system testing 90 of these were detected.
- 150 other errors were also detected
- **How many unknown errors remain after system testing?**

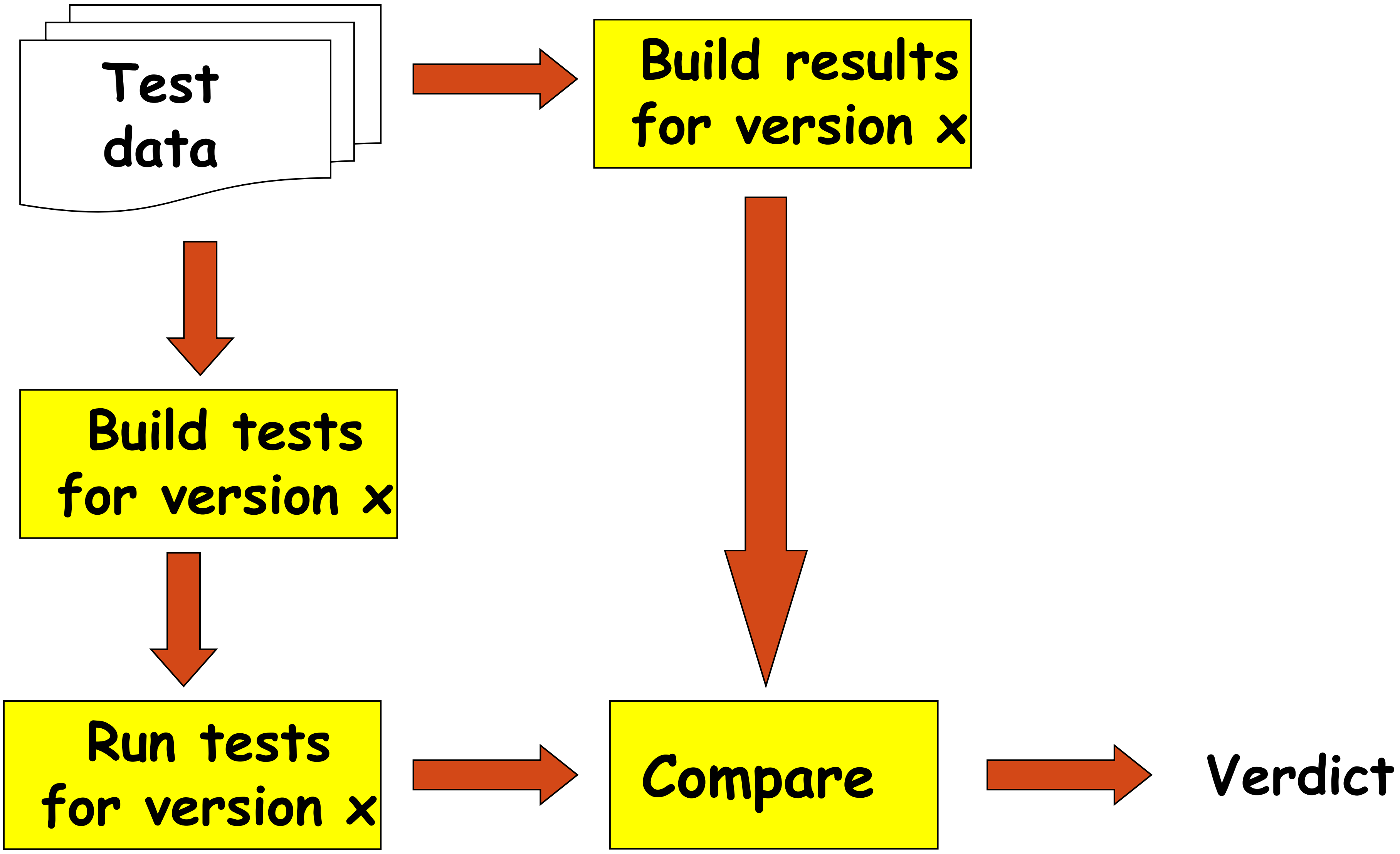
Regression testing

- Regression testing is testing done to check that a system update does not cause new errors or re-introduce errors that have been corrected earlier.
- It spans unit, integration, and system level testing
- Any system during use undergoes frequent code changes.
  - Corrective, Adaptive, and Perfective changes.
- Regression testing needed after every change:
  - Ensures unchanged features continue to work fine.
- Resolution testing checks whether the defect has been fixed. Regression testing checks whether the unmodified functionalities still continue to work correctly.

Major Regression Testing Tasks

- Test revalidation (RTV):
  - Check which tests remain valid
- Test selection (RTS):
  - Identify tests that execute modified portions.
- Test minimization (RTM):
  - Remove redundant tests.
- Test prioritization (RTP):
  - Prioritize tests based on certain criteria.

Automating regression testing



- **End of Chapter**

Thanks