

Software Testing and Quality Assurance

Unit-2 Levels of Testing



Unit-2 Levels of Testing

2.0. Unit Testing

2.1. Control Flow Testing

2.2. Data Flow Testing

2.3. Domain Testing

2.4. System Integration Testing

2.5. System Testing

2.6. Acceptance Testing



Unit Testing

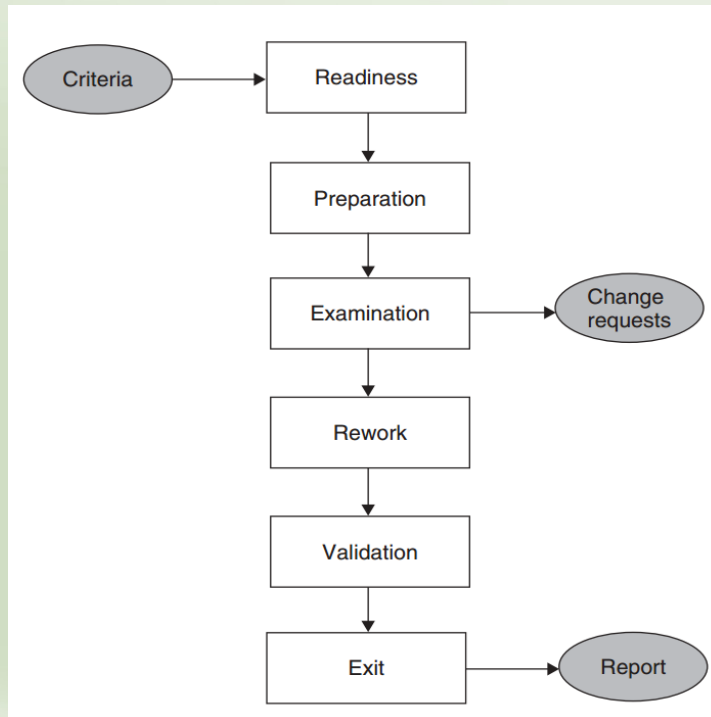
- ❑ **Unit Testing:** Refers to testing program units in isolation.
- ❑ **Units** are **functions, procedures, or methods**, even a **class**.
- ❑ **Unit Testing** removes **dependencies** on other program units and helps to verify the unit alone produce desirable result.
- ❑ Intuitively, a programmer needs to test a unit via
 - **Execute every line of code**.
 - **Execute every predicate** in the unit to evaluate them as true and false separately.
 - Observe that the unit performs its intended function and ensure that it contains **no known errors**.
- **Unit Testing** is conducted in two complementary phases:
 - **Static unit Testing:** is a **non-execution** based via **code review**
 - **Dynamic unit Testing:** is **execution based**.



Unit Testing

□ In Static Unit Testing:

In static unit testing, code is reviewed by applying techniques commonly known as **inspection** and **walkthrough**.



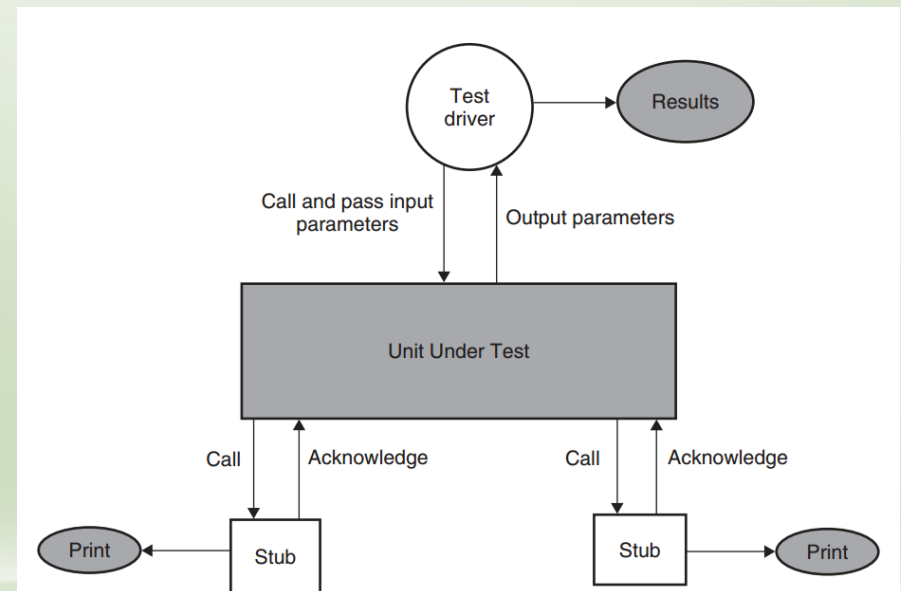
Unit Testing

❑ In Dynamic Unit Testing

- In this testing, a program unit is **actually executed in isolation**
- Dynamic unit testing is created by **emulating the context** of the unit under test
- The context of a unit test consists of two parts: (i) **a caller of the unit (Test Driver)** and (ii) all the units **called by the unit (Stub)**.

Test Driver: is a program that invokes the unit under test with input values.

Stubs: A stub is a “dummy subprogram” that replaces a unit that is called by the unit under test.



Unit Testing

❑ Defect Prevention: include suitable mechanism in the code

- **Built in tracking, tracing mechanisms**
- Include **exceptions** such as **division by zero, array index out of bounds, buffer overflow or underflow, File Not found, db connection error** and etc.

❑ Debugging: The process of determining the cause of a program failure and fixing it.

In Unit Testing modules contains the following

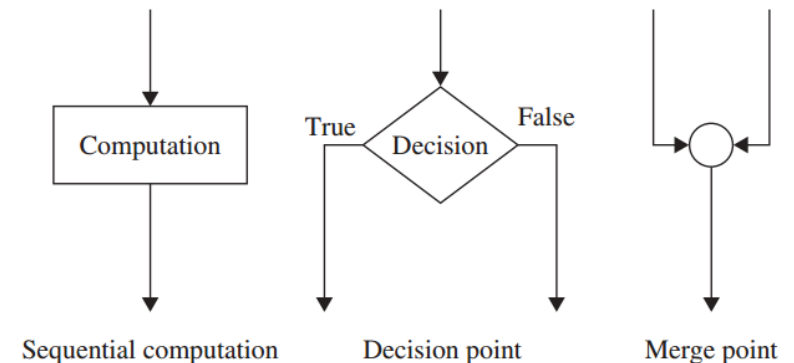
- Sequence of statements



2.1. Control flow testing

Control Flow Testing :

- ❑ **Control flow** refers to a code that define **sequence**, **decision**, **function**, **class** and or etc of a program.
- ❑ Control flow testing is a kind of **structural testing**, which is performed by **programmers** to test code written by them.
- ❑ The main idea in **control flow** testing is to appropriately select **few paths** in a program unit and observe whether or not the selected paths produce the expected outcome.
- ❑ A **control flow graph(CFG)** is a graphical representation of a program unit with.
 - **Sequence**, **function**, **loop**, **decision**, **Flow direction (arrow)** and others as call



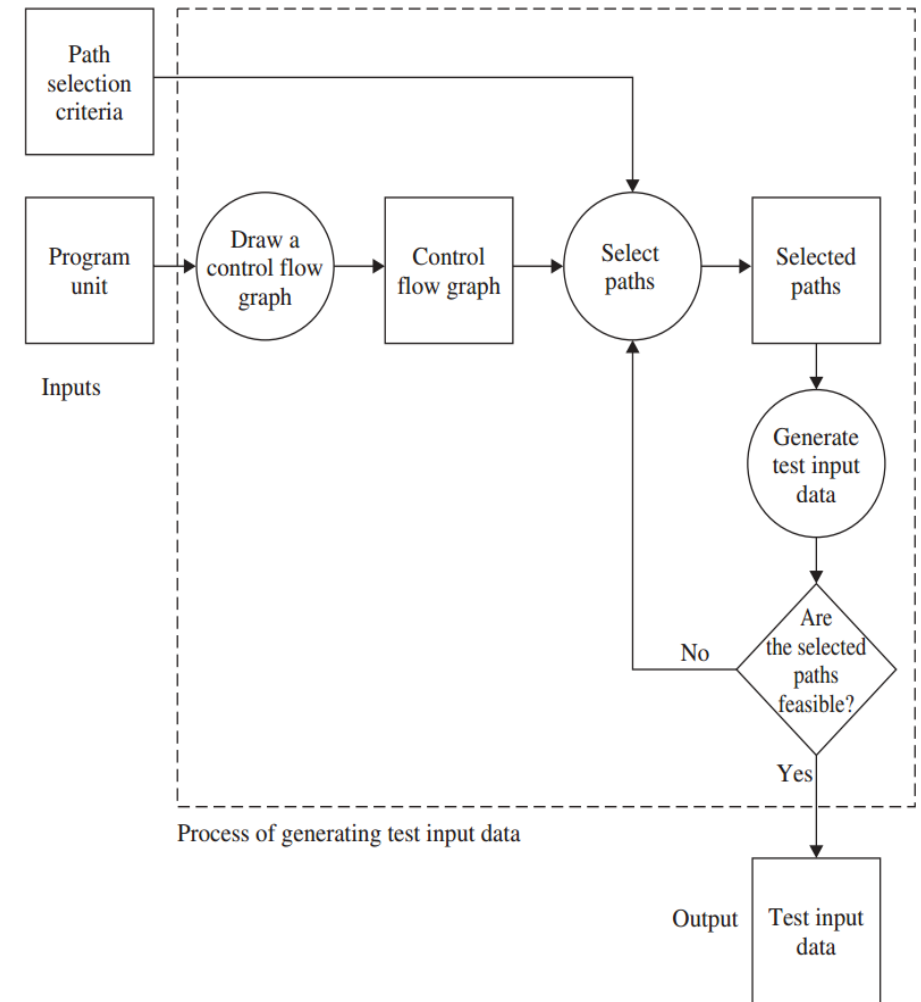
2.1. Control flow testing

- ❑ The process of generating test input data for control flow testing

Input : source code, path selection criteria

CFG: visualize all paths

Selection of Paths: Paths are selected by considering the structure of the CFG.



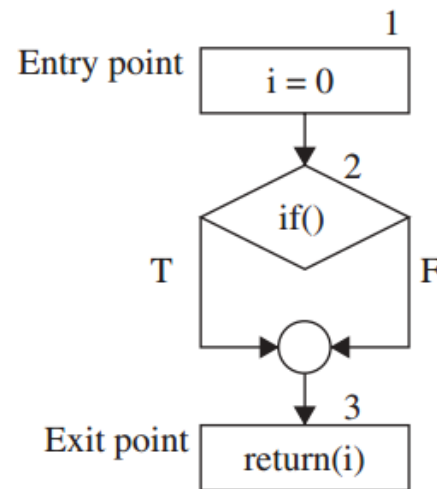
2.1. Control flow testing

Program unit for openfiles()

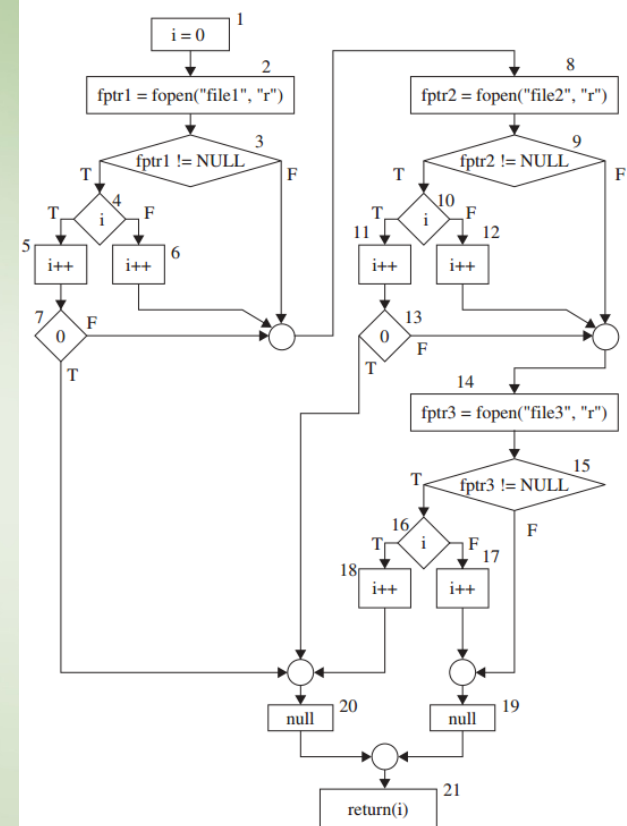
```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
    /*
     * This function tries to open files "file1", "file2", and
     * "file3" for read access, and returns the number of files
     * successfully opened. The file pointers of the opened files
     * are put in the global variables.
     */
    int i = 0;
    if(
        ((( fptr1 = fopen("file1", "r")) != NULL) && (i++)
          && (0)) ||
        ((( fptr2 = fopen("file2", "r")) != NULL) && (i++)
          && (0)) ||
        ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
    );
    return(i);
}
```

- How many paths? How many decision ?
sequence ? assignment ?



High-Level CFG
representation of openfiles()



Detailed CFG representation of
openfiles()

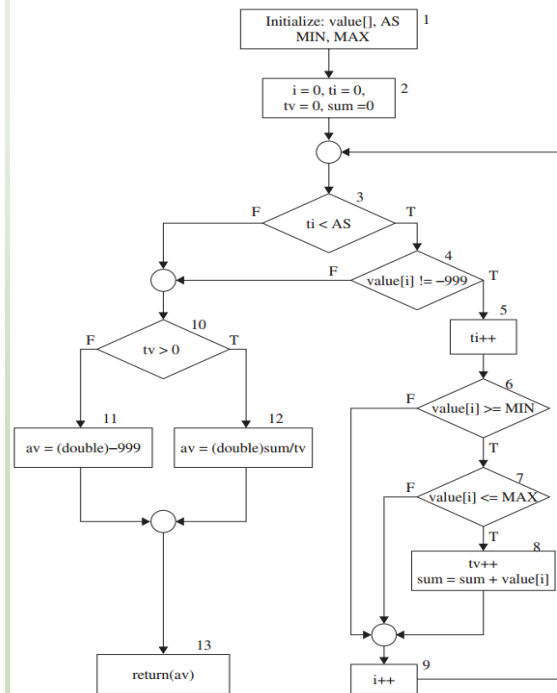


2.1. Control flow testing

- **Paths in a Control Flow Graph**
- A path is represented as a **sequence of computation and decision nodes** from the **entry node** to **the exit node**.

```
public static double ReturnAverage(int value[],
                                   int AS, int MIN, int MAX){
    /*
    Function: ReturnAverage Computes the average
    of all those numbers in the input array in
    the positive range [MIN, MAX]. The maximum
    size of the array is AS. But, the array size
    could be smaller than AS in which case the end
    of input is represented by -999.
    */
    int i, ti, tv, sum;
    double av;
    i = 0; ti = 0; tv = 0; sum = 0;
    while (ti < AS && value[i] != -999) {
        ti++;
        if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
        }
        i++;
    }
    if (tv > 0)
        av = (double)sum/tv;
    else
        av = (double) -999;
    return (av);
}
```

Function to compute average of an array.



CFG Function to compute the average of an array.

Path 1	1-2-3(F)-10(T)-12-13
Path 2	1-2-3(F)-10(F)-11-13
Path 3	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
Path 4	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Paths in compute average

Path selection criteria:

- Select **all paths**.
- Select paths to achieve complete **statement coverage**.
- Select paths to achieve complete **branch coverage**.
- Select paths to achieve **predicate coverage**.



2.1. Control flow testing

All Path coverage criteria

- If *all* the paths in a CFG are selected, then one can detect all faults
- The `openfiles()` function shown above contains more than 25 paths and only eight of all those paths are feasible
- If one selects all possible paths in a program, then we say that the *all-path* selection criterion has been satisfied.

Example of the `openfiles()` function tries to open the three files **file1**, **file2**, and **file3**. A file is said to be successfully opened with “read” access if the **file exists**. The existence of a file is either “yes” or “no.” Thus, the input domain of the function consists of eight combinations of the existence of the three files.

Existence of file1	Existence of file2	Existence of file3
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

Input Domain of `openfiles()`

Input	Path
< No, No, No >	1-2-3(F)-8-9(F)-14-15(F)-19-21
< Yes, No, No >	1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21
< Yes, Yes, Yes >	1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14-15(T)-16(T)-18-20-21

Example for three Inputs and Paths in `openfiles()`

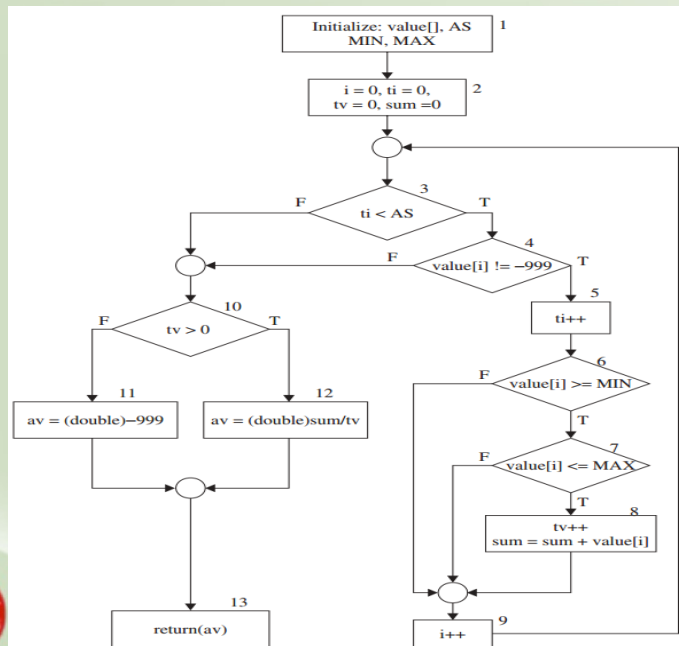


2.1. Control flow testing

Statement Coverage Criterion

- Refers to executing **individual program statements** and observing the outcome
- Complete statement coverage is the **weakest coverage** criterion in program testing.
- Any test suite that achieves **less than statement coverage** for new software is considered to be **unacceptable**.

The **ReturnAverage()** method example



```
i    = 0;
ti   = 0;
tv   = 0;
sum  = 0;
```

Node 2

```
(ti < AS && value[i] != -999)
```

Node 3 and Node 4

SCPath 1 1-2-3(F)-10(F)-11-13

SCPath 2 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

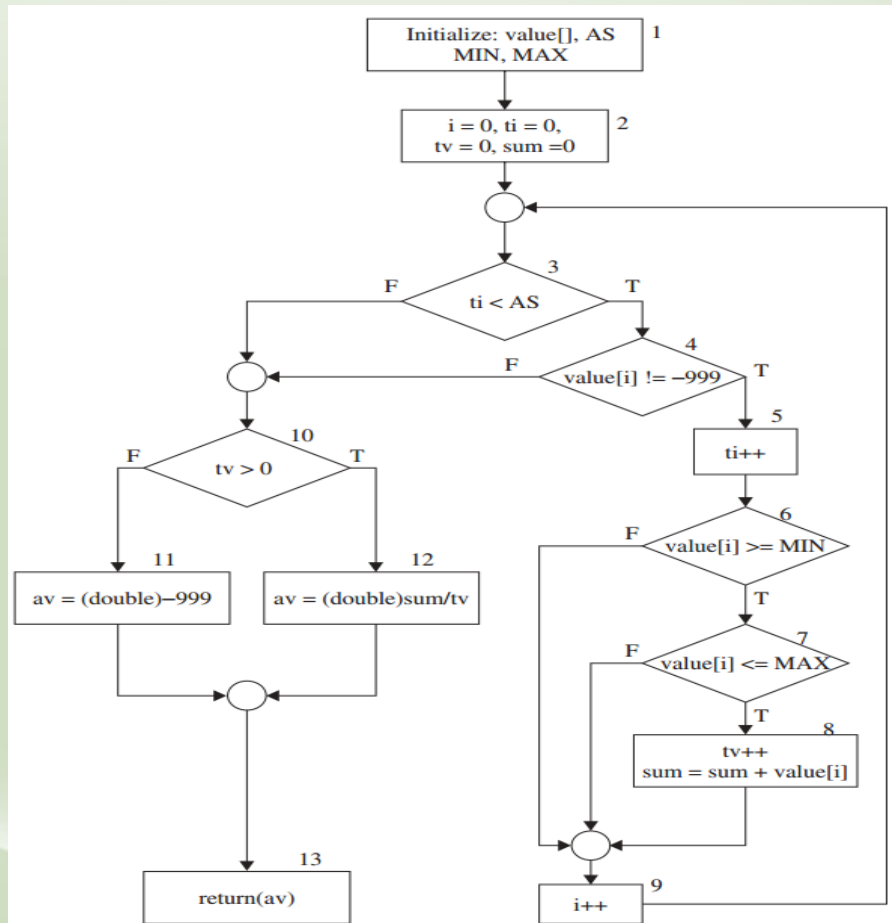
Paths for Statement Coverage of CFG compute Average



2.1. Control flow testing

Branch Coverage Criterion

- Covering a branch means selecting a path that includes the outgoing edge



BCPath 1	1-2-3(F)-10(F)-11-13
BCPath 2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
BCPath 3	1-2-3(T)-4(F)-10(F)-11-13
BCPath 4	1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13
BCPath 5	1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13

Paths for Branch Coverage of CFG Compute average

Predicate Coverage Criterion

- If all possible combinations of **truth values** of the conditions affecting a selected path have been explored under some tests, then we say that **predicate coverage** is satisfied.



Reading

- Generating Test Input
- What is Infeasible Path



2.2. Data flow testing

□ Testing paths that include pairs of **definition** and **the use of variables** in programs

Data Flow Anomaly:

- An abnormal situation to successively assign two values to a variable without using the first value a.k.a **Defined and then Defined Again**.
- It is abnormal to use a value of a variable before assigning a value to the variable. a.k.a **Undefined but Referenced**.
- Abnormal situation is to generate a data value and never use it a.k.a **Defined but Not Referenced**.



2.2. Data flow testing

Data flow graph

- ❑ A **data flow graph** is drawn with the objective of identifying **data definitions** and their **uses**.
- Each occurrence of a data variable is classified as
 - **Definition**: This occurs when a value is moved into the memory location of the variable.
 - **Undefined or Kill**: This occurs when the value and the location become unbound.
 - **Use**: This occurs when the value is fetched from the memory location of the variable.
 - **Computation use(c-use)**: to produce new variable value or the same variable of different value.
 - **Predicate use (p-use)**: controlling the flow of execution.
- A **data flow graph** is a **directed graph** constructed as follows:
 - A sequence of **definitions** and **c-uses** is associated with each **node** of the graph.
 - A set of **p-uses** is associated with each **edge** of the graph.
 - The entry node has a definition of each parameter and each nonlocal variable which occurs in the subprogram.
 - The **exit** node has an **undefined** of each local variable

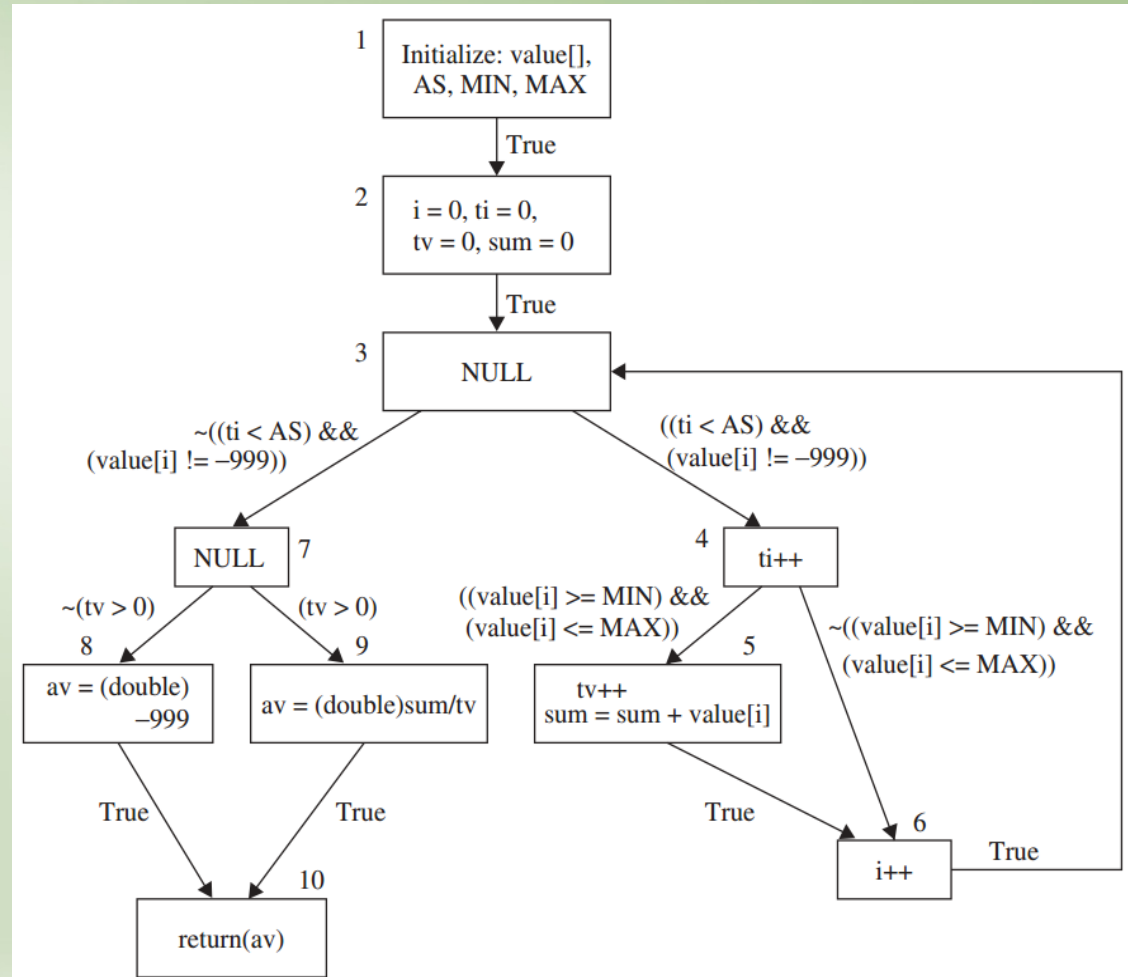


2.2. Data flow testing

- **Node 1**, represents initialization of the input vector.
- **Node 2**, represents the initialization of the four local variables
- **Node 3**, NULL node indicator of the control exiting from the loop,
- The predicate associated with **edge (3, 4)** is the condition part of the while loop
-
-
- Finally, the return(av) statement is represented by **node 10**.

Data Flow Terms

- A **variable defined** in a statement is used in another statement which may occur immediately or several statements after the definition.



Data flow graph of ReturnAverage() example.



2.2. Data flow testing

Data Flow Terms:

- We are interested in **finding paths** that include pairs of **definition** and the **use** of variables.
- **Global c-use**: A c-use of a variable x in node i is said to be a global c-use if x has been defined before in a node other than node i .
 - The **c-use** of variable tv in node **9** is a global c-use since tv has been defined in nodes **2** and **5** in the above DFG.
- **Definition Clear Path**: A path $(i - n_1 \dots n_m - j)$, $m \geq 0$, is called a definition clear path (def-clear path) with respect to variable x
 - from node i to node j and
 - from node i to edge (n_m, j)
- For example The paths **2-3-4-5** and **2-3-4-6** are def-clear paths with respect to variable tv from node **2** to **5** and from node **2** to **6**, respectively.
- **Global Definition**: A node i has a global definition of a variable x if node i has a definition of x and there is a def-clear path with respect to x from node i to some 1, node containing a global c-use or 2, edge containing a p-use of



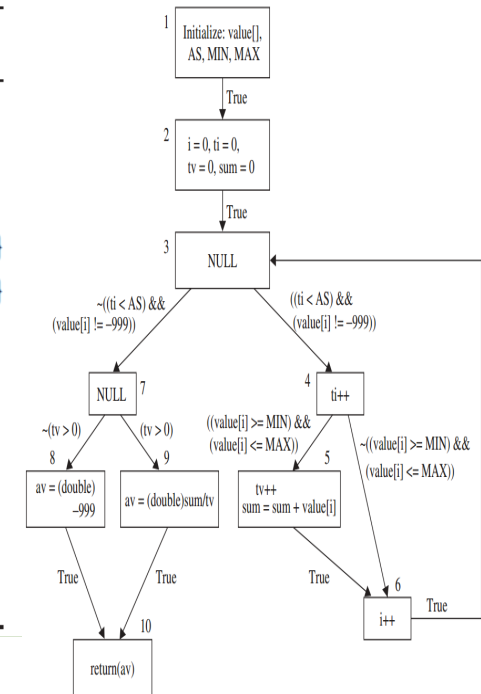
2.2. Data flow testing

- **Simple Path:** is a path in which all nodes, except possibly the first and the last, are distinct.
- $\text{def}(i)$ denotes the set of variables which have global definitions in node i . Similarly, $\text{c-use}(i)$ denotes the set of variables which have global c-uses in node i .

Nodes i	$\text{def}(i)$	$\text{c-use}(i)$	Edges (i, j)	$\text{predicate}(i, j)$	$\text{p-use}(i, j)$
1	{value, AS, MIN, MAX}	{}	(1, 2)	True	{}
2	{i, ti, tv, sum}	{}	(2, 3)	True	{}
3	{}	{}	(3, 4)	$(ti < AS) \ \&\& \ (value[i] \neq -999)$	{i, ti, AS, value}
4	{ti}	{ti}	(4, 5)	$(value[i] \leq MIN) \ \&\& \ (value[i] \geq MAX)$	{i, MIN, MAX, value}
5	{tv, sum}	{tv, i, sum, value}	(4, 6)	$\sim((value[i] \leq MIN) \ \&\& \ (value[i] \geq MAX))$	{i, MIN, MAX, value}
6	{i}	{i}	(5, 6)	True	{}
7	{}	{}	(6, 3)	True	{}
8	{av}	{}	(3, 7)	$\sim((ti < AS) \ \&\& \ (value[i] \neq -999))$	{i, ti, AS, value}
9	{av}	{sum, tv}	(7, 8)	$\sim(tv > 0)$	{tv}
10	{}	{av}	(7, 9)	$(tv > 0)$	{tv}
			(8, 10)	True	{}
			(9, 10)	True	{}

Global definitions and global c-uses appearing in the **DFG**

Predicates and p-uses appearing in the data flow graph



- **Du-path:** A path $(n_1 - n_2 - \dots - n_j - n_k)$ is a **definition-use path** (du-path) with respect to (w.r.t) variable x if node n_1 has a global definition of x and either node n_k has a global c-use of x and $(n_1 - n_2 - \dots - n_j - n_k)$ is a def-clear simple path w.r.t. x or edge (n_j, n_k) has a p-use of x and $(n_1 - n_2 - \dots - n_j)$ is a def-clear, loop-free path w.r.t. x .



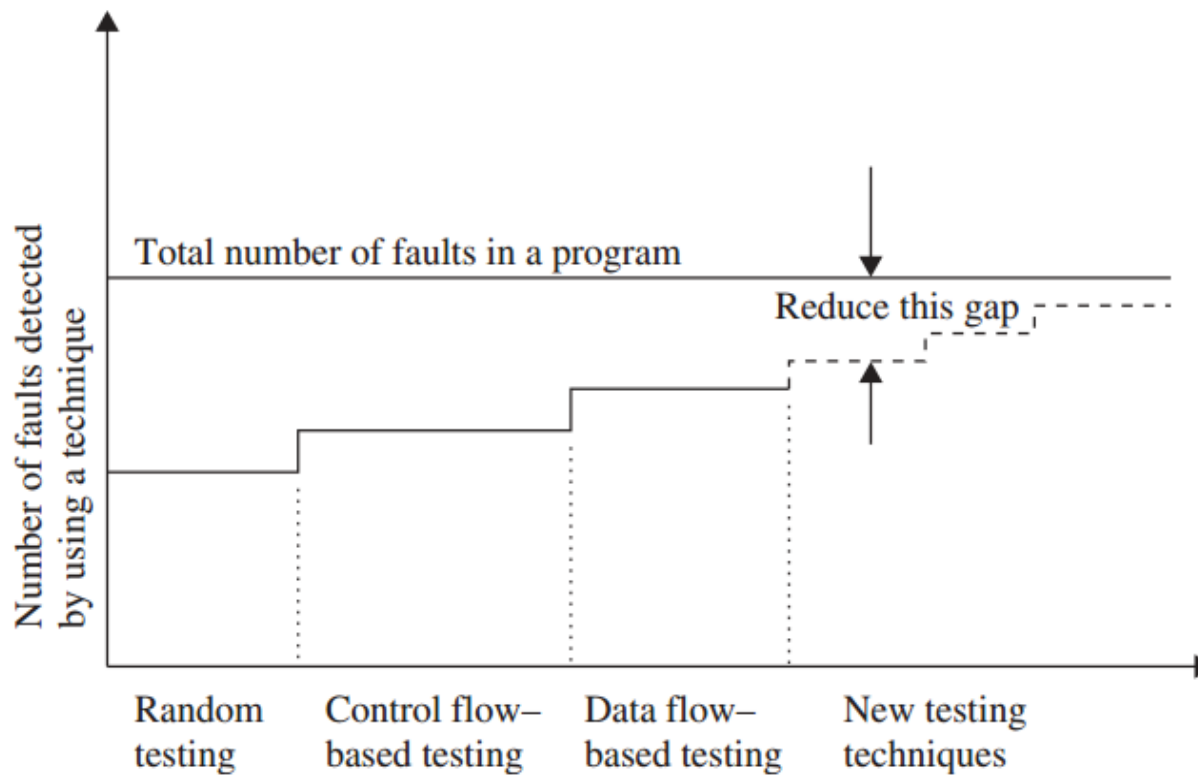
2.2. Data flow testing

- There are **seven data flow testing criterion** based on **definitions** and **uses** —both **c-uses** and **p-uses**—of variables.
 1. **All-defs**: For each variable x and for each node i such that x has a global definition in node i , select a complete path which includes a def-clear path from node i to
 - node j having a global c-use of x or
 - edge (j, k) having a p-use of x .
 2. **All-c-uses**: For each variable x and for each node i , such that x has a global definition in node i , select complete paths which include def-clear paths from node i to all nodes j such that there is a global c-use of x in j .
 3. **All-p-uses**: For each variable x and for each node i such that x has a global definition in node i , select complete paths which include def-clear paths from node i to all edges (j, k) such that there is a p-use of x on edge (j, k) .
 4. **All-p-uses/Some-c-uses**
 5. **All-c-uses/Some-p-uses**
 6. **All-uses**
 7. **All-du-paths**



2.2. Data flow testing

Limitation of different fault detection techniques.



2.2. Data flow testing

Self-assessment questions

1. Compare
 - random test selection,
 - test selection based on control flow, and
 - test selection based on data flow.
2. Explain feasible testing paths, and feasible path selection criterion with concrete example.



2.3. Domain Testing

- **Two fundamental elements** of a computer program are **input domain** and **program paths**.
- **The input domain** of a program is the set of **all input data** to the program.
- **A program path** is a sequence of instructions **from the start of the program** to **some point of interest in the program**.
- A path is said to be **feasible** if **there exists an input data** which causes the program to execute the path. Otherwise, the path is said to be **infeasible**.

Two common domain errors

- **Computation Error:** A computation error occurs when a specific input data causes the program to execute the correct, i.e., desired path, but the output value is wrong. e.g. function result = $f(a, b)$ is executed as result = $f(b, a)$.
- **Domain Error:** An incorrect path can be selected by a program if there is a fault in one or more of the conditional statements in the program. e.g. when p is wrong in `if (p) then f1() else f2()`, a wrong function will be called.



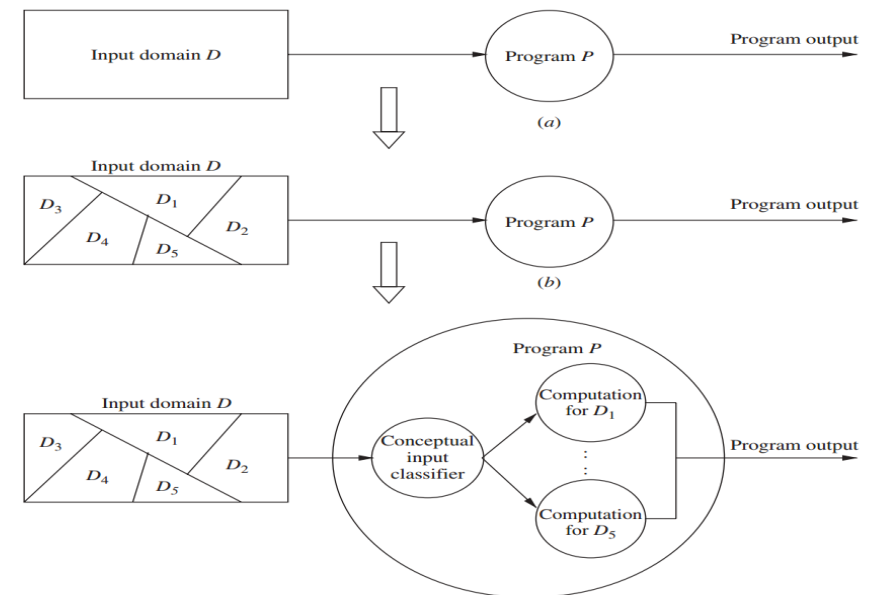
2.3. Domain Testing

- A **domain** is a set of **input values** for which the program performs the same computation for every member of the set. We are interested in **maximal domains** such that the program performs **different computations** on **adjacent domains**.
- A program is said to have a domain error if the program incorrectly performs input classification. Assuming that adjacent domains perform different computations, a domain error will cause the program to produce incorrect output.

(a) The set D is the entire input set of a program P

(b) P performs five different computations, one for each subset $D_1, \dots, D_5$.

(c) Five different computations, computation for D_1 through computation for D_5 , for subsets $D_1, \dots, D_5$, respectively



2.3. Domain Testing

Testing For Domain Errors

- The difference between **flow graph-based** (control and data flow graph) testing techniques and **domain testing**.
 - **Select paths** from a control flow graph or a data flow graph **to satisfy certain coverage criteria**.
 - **Domain testing** defines a category of **faults**, called *domain errors*, and selects test data to detect those faults.

Source of Domains :

- Domains can be identified from both **specifications** and **programs** using the steps.
 - Draw a control flow graph from the given **source code**.
 - Find all possible **interpretations** of the **predicates**.
 - Analyze the interpreted **predicates** to identify **domains**.



2.3. Domain Testing

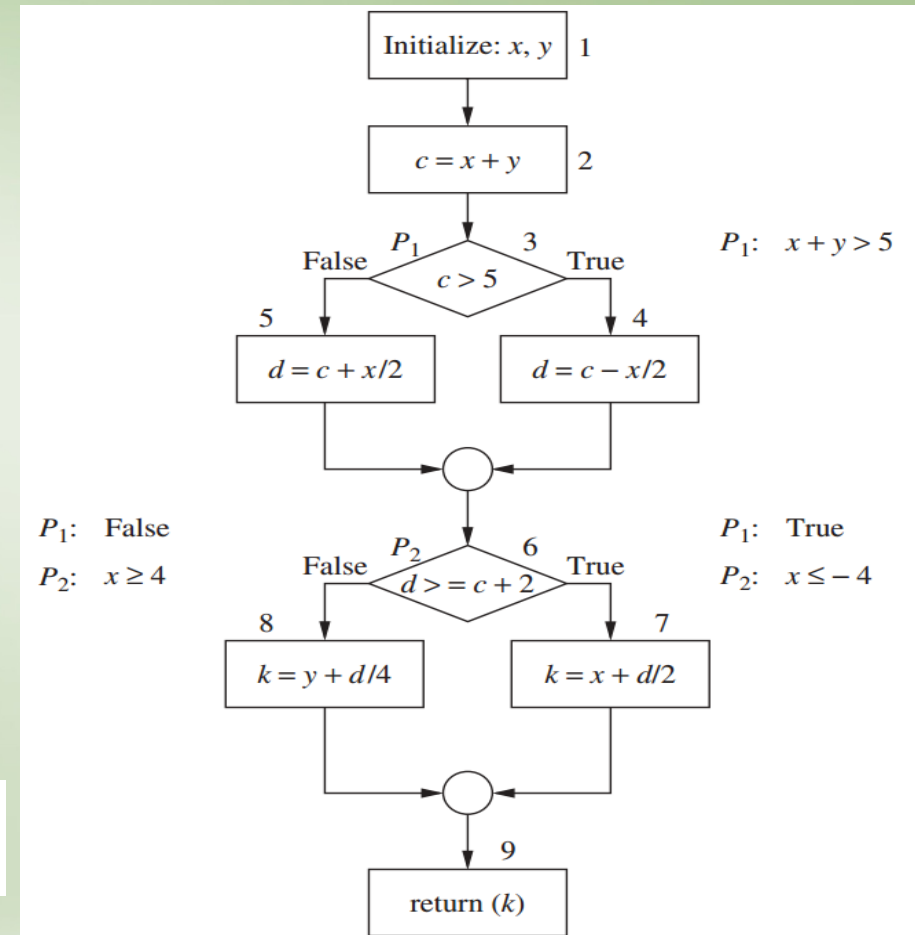
```
int codedomain(int x, int y){  
    int c, d, k  
    c = x + y;  
    if (c > 5) d = c - x/2;  
    else      d = c + x/2;  
    if (d >= c + 2) k = x + d/2;  
    else      k = y + d/4;  
    return(k);  
}
```

A function to explain program domains.

The first **if()** statement has just one interpretation

$P_1: x + y > 5$

The second **if()** statement gets two interpretations $P_1 = \text{True}$ and $P_1 = \text{False}$ leads us to P_2



2.3. Domain Testing

Domains and its Boundaries:

- **Closed Boundary:** A boundary is said to be closed if the points on the boundary are included in the domain of interest. Eg: $P_1: x \leq -4$
- **Open Boundary:** A boundary is said to be open if the points on the boundary do not belong to the domain of interest. Eg: $P_1: x+y > 4$
- **Closed Domain:** A domain is said to be closed if all of its boundaries are closed.
- **Open Domain:** A domain is said to be open if some of its boundaries are open.
- **Extreme Point:** An extreme point is a point where two or more boundaries cross.
- **Adjacent Domains:** Two domains are said to be adjacent if they have a boundary inequality in common.



2.3. Domain Testing

Types of Domain Errors:

1. **Closure Error:** A closure error occurs if a boundary is open when the intention is to have a closed boundary, or vice versa. e.g. **The operator $\leq$ is implemented as $<$ or vice versa**
2. **Shifted-Boundary Error:** A shifted-boundary error occurs when the implemented boundary is parallel to the intended boundary. **e.g. $P1:x+y>5$ is used instead of $P1:x+y>4$.**
3. **Tilted-Boundary Error:** If the constant coefficients of the variables in a predicate defining a boundary take up wrong values, then the tilted-boundary error occurs. e.g. **$P1:x+y>5$ instead of $P1 : x + 0.5y > 5$.**



2.3. Domain Testing

Test selection Criterion

1. Closed inequality boundary
 - a. Boundary shift resulting in a reduced domain
 - b. Boundary shift resulting in an enlarged domain
 - c. Boundary tilt
 - d. Closure error
2. Open inequality boundary
 - a. Boundary shift resulting in a reduced domain
 - b. Boundary shift resulting in an enlarged domain
 - c. Boundary tilt
 - d. Closure error
3. Equality boundary



2.3. Domain Testing

Self assessments

1. The domain testing strategy requires us to select test points on and/or very close to domain boundaries. Why don't we select test points far from the boundaries?
2. Explain the difference between control flow-based testing and domain error-based testing
3. explain the test selection criterion in domain-based testing and show the closed inequality error (boundary shift resulting in a reduced domain) is detected by the test points chosen by the selection criterion
4. Identify some difficulties in applying the concept of domain testing to actual program testing.



2.4. System Integration Testing

- A software module, or component, is a self-contained element of a system. Modules have well-defined interfaces with other modules.
- A system is a collection of modules interconnected in a certain way to accomplish a tangible objective.
- system integration testing is a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing.
- Integration testing is said to be complete when the system is fully integrated together, all the test cases have been executed, all the severe and moderate defects found have been fixed, and the system is retested



2.4. System Integration Testing

Different Types Of Interfaces

- Modules are interfaced with other modules to realize the system's functional requirements.
 - **Procedure Call Interface** the caller passes on control to the called module.
 - **Shared Memory** A block of memory is shared between two modules.
 - **Message Passing Interface:** One module prepares a message by initializing the fields of a data structure and sending the message to another module.

Interface Errors

- Construction
- Inadequate Functionality
- Location of Functionality
- Added Functionality
- Misuse of Interface
- Misunderstanding of Interface and etc...



2.4. System Integration Testing

The major advantages of conducting system integration testing are as follows:

- Defects are detected early.
- It is easier to fix defects detected earlier.
- We get earlier feedback on the health and acceptability of the individual modules and on the overall system.
- Scheduling of defect fixes is flexible, and it can overlap with development.

Granularity of System Integration Testing

- **Intra-system testing** combining the modules together to build a cohesive system.
- **Inter-system testing** Integrating a call control system and a billing system in a telephone network is another example of intersystem testing.
- **Pairwise system testing** For example, in testing communication between a network element (radio node) and the element management systems



2.4. System Integration Testing

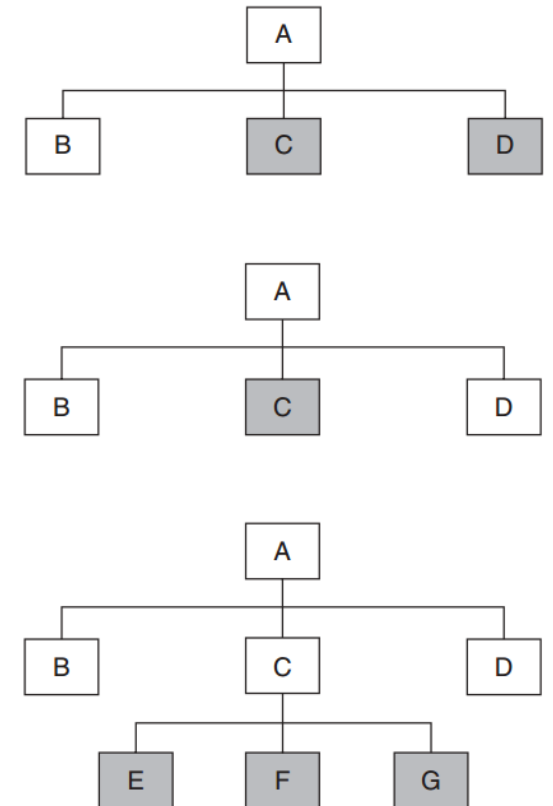
System Integration Techniques:

- **Incremental** a series of test cycles created to add new modules in existing build.
- **Top down testing** top level modules are tested and down in the hierarchy.

Top-down integration of modules A and B
C and D are test drivers

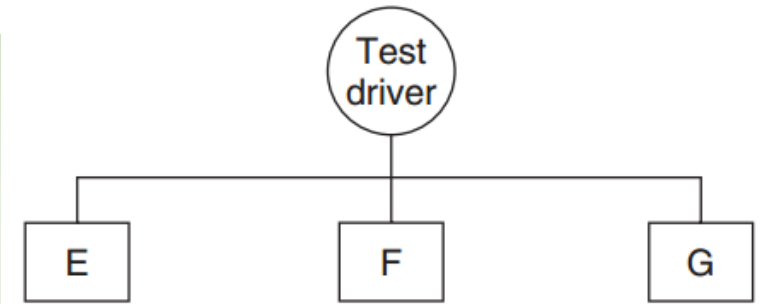
Top-down integration of modules A, B, and D

Top-down integration of modules A, B,
D, and C



2.4. System Integration Testing

Bottom Up integration begins with the integration of lowest level modules that invokes no other modules.



Bottom-up integration of modules E, F, and G

Sandwich and Big Bang

- In **Sandwich** mix of the top-down and bottom-up approaches.
- In **Big Bang** first, all modules are individually tested. then, all those modules are put together to construct the entire system which is also tested as a whole.



Compare the Top-Down and Bottom-Up approaches

- **Validation of Major Design Decisions:** The top-level modules contain major design decisions. Faults in design decisions are detected early if integration is done in a top-down manner. In the bottom-up approach, those faults are detected toward the end of the integration process.
- **Observation of System-Level Functions:** One applies test inputs to the top-level module, which is similar to performing system-level tests in a very limited way in the top-down approach. This gives an opportunity to the SIT personnel and the development team to observe system-level functions early in the integration process. However, similar observations can be done in the bottom-up approach only at the end of system integration.
- **Difficulty in Designing Test Cases:** In the top-down approach, as more and more modules are integrated and stubs lie farther away from the top-level module, it becomes increasingly difficult to design stub behavior and test input. This is because stubs return predetermined values, and a test engineer must compute those values for a given test input at the top level. However, in the bottom-up approach, one designs the behavior of
- **Reusability of Test Cases:** In the top-down approach, test cases designed to test the interface of a newly integrated module is reused in performing regression tests in the following iteration. Those test cases are reused as system-level test cases. However, in the bottom-up approach, all the test cases incorporated into test drivers, except for the top-level test driver, cannot be reused. The top-down approach saves resources in the form of time and money.



2.4. System Integration Testing

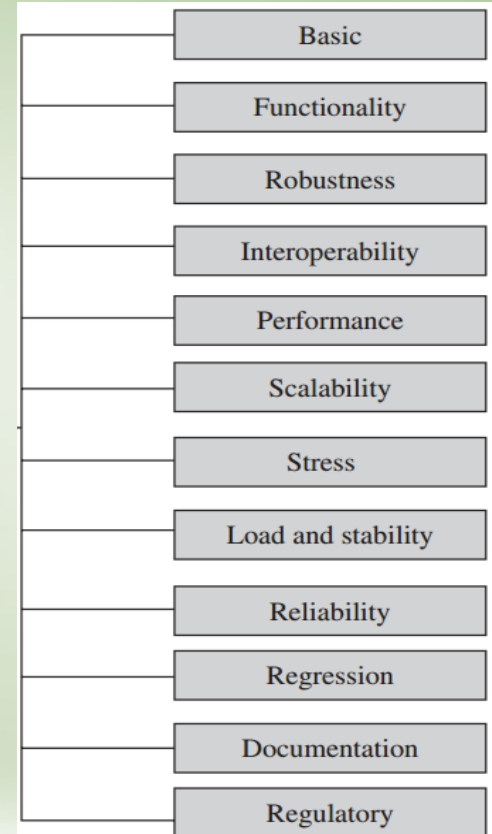
Self Assessment

1. Describe the difference between black-box and white-box testing techniques?
2. If a program passes all the black-box tests, it means that the program should work properly. Then, in addition to black-box testing, why do you need to perform white-box testing?
3. Describe the difference between unit testing and integration testing? Why should integration testing be performed? What types of errors can this phase of testing reveal?
4. Discuss the advantages and disadvantages of top-down and bottom-up approaches to integration testing.
5. Does automation of integration tests help the verification of the daily build process? Justify your answer.



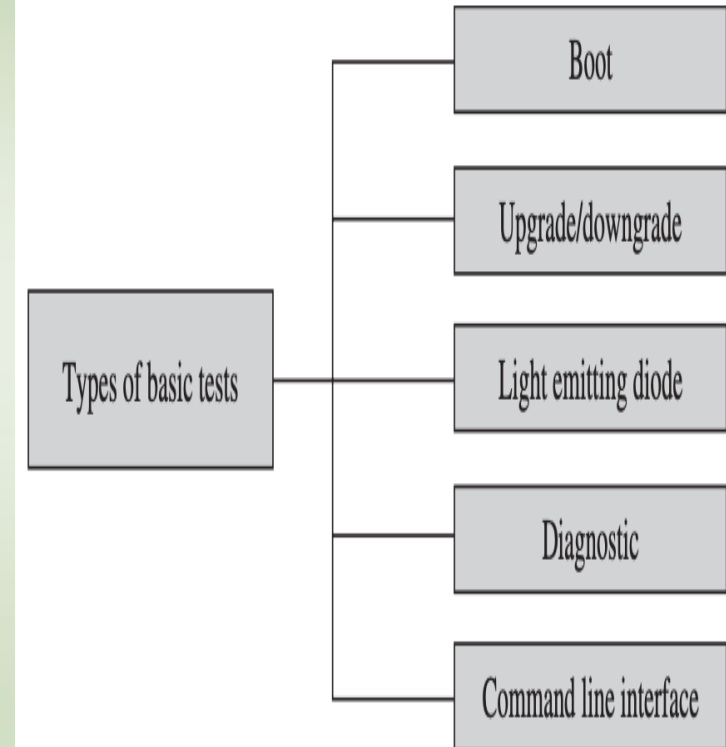
2.5. System Testing

- The objective of system-level testing is to establish whether an **implementation conforms** to **the requirements specified by the customers**.
- **Basic tests** provide an evidence that the system can be **installed**, configured, and brought to an operational state.
- **Functionality tests** provide comprehensive testing of requirements within the **capabilities of the system**.
- **Robustness tests** determine **how well the system recovers** from **various input errors and other failure situations**.
- **Interoperability tests** determine whether the system can **interoperate with other third-party products**.
- **Performance tests** measure the **throughput** and **response time**, under **various conditions**.
- **Scalability tests** determine **user scaling**, **geographic scaling**, and **resource scaling**.
- **Stress tests** put a system under stress in order **to determine the limitations of a system** and, **when it fails**, to determine **the manner in which the failure occurs**.
- **Regression tests** determine that the system remains stable **as it cycles** through the integration of other subsystems and through maintenance tasks.



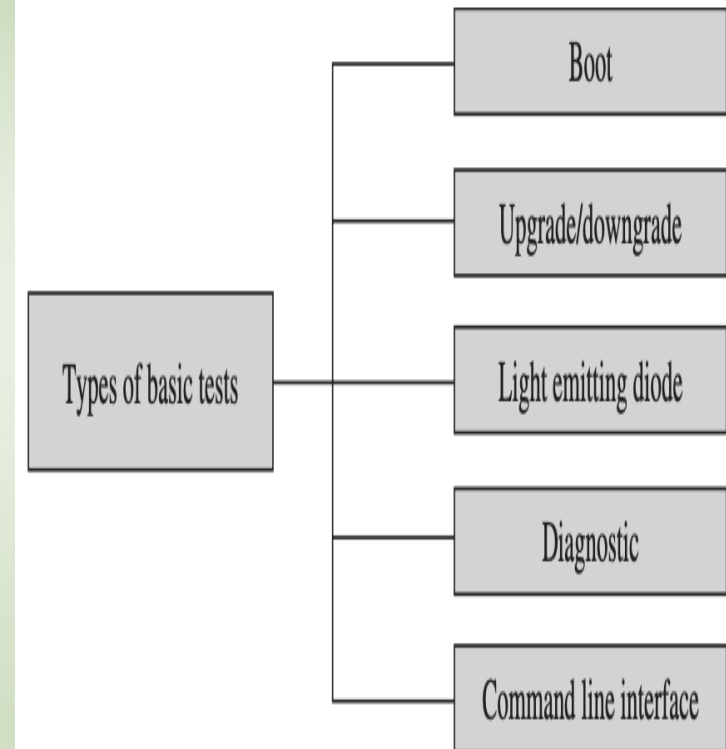
Basic Tests

- ❑ The basic tests give evidence that the system is **ready for more rigorous tests**.
- ❑ The objective is to establish that there is sufficient evidence that a system can **operate without trying to perform thorough testing**.
- ❑ **Basic tests** are performed to ensure that commonly used functions, not all of which may directly relate to user-level functions, work to our satisfaction.
- ❑ **Boot tests** are designed to verify that the system can boot up its software image (or build) from the supported boot options. The boot options include booting from **ROM**, **FLASH card**, and **PCMCIA** (Personal Computer Memory Card International Association) card.
- ❑ **Upgrade/downgrade tests** are designed to verify that the system software can be upgraded or downgraded (rollback) in a graceful manner from the previous version to the current version or vice versa.
- ❑ An upgradation process may fail because of a number of different conditions: **user-invoked abort** (the user interrupts the upgrade process), **in-process network disruption** (the network environment goes down), **in-process system reboot** (there is a power glitch), or **self-detection of upgrade failure** (this is due to such things as insufficient disk space and version incompatibilities)

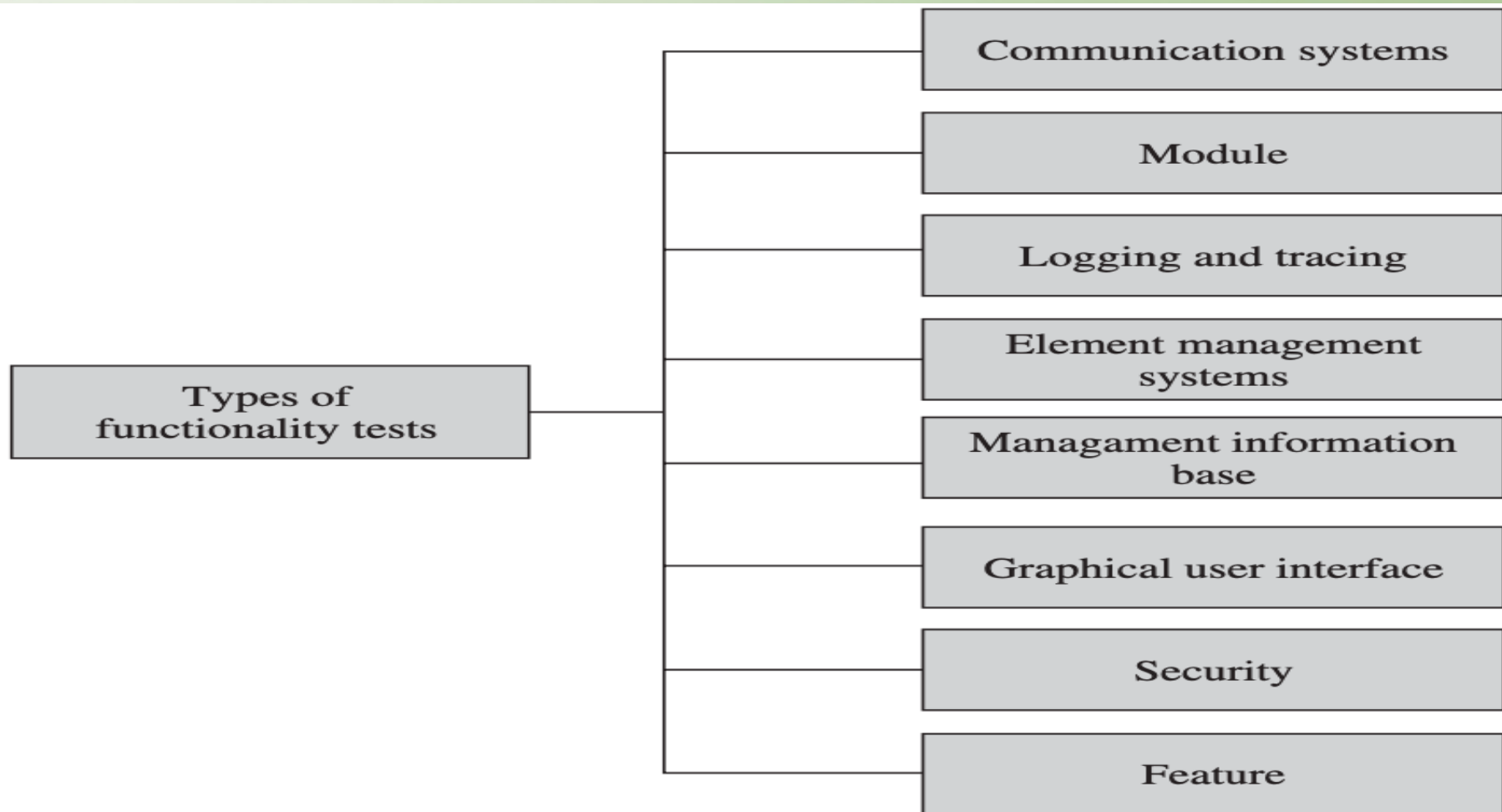


Basic Tests

- ❑ The LED (light emitting diode) tests are designed to verify that the system **LED status indicators function as desired**.
- ❑ The LED tests are designed to ensure that the visual operational status of the system and the submodules are correct. Examples of LED tests at the system and subsystem levels are as follows:
 - System LED test: green = OK, blinking green = fault, off = no power.
 - Ethernet link LED test: green=OK, blinking green=activity, off=fault.
 - Cable link LED test: green = OK, blinking green = activity, off = fault.
 - User-defined T1 line card LED test: green=OK, blinking green=activity, red = fault, off = no power.
- ❑ **Diagnostic tests** are designed to verify that the hardware components (or modules) of the system are functioning as desired.
- ❑ Diagnostic tests monitor, isolate, and identify system problems without manual troubleshooting. Some examples of diagnostic tests are as follows:
- ❑ **Power-On Self-Test(POST), Ethernet Loop-Back Test, Bit Error Test (BERT).**
- ❑ **The CLI tests** are designed to verify that the system can be configured, or provisioned, in specific ways.



FUNCTIONALITY TESTS



FUNCTIONALITY TESTS

- ❑ **Communication Systems:** Communication systems tests are designed to verify the implementation of the communication systems as specified in the customer requirements specification.
- ❑ **Module:** Module tests are designed to verify that all the modules function individually as desired within the systems.
- ❑ **Logging and Tracing:** Logging and tracing tests are designed to verify the configurations and operations of logging and tracing. T
- ❑ **Element Management:** The EMS tests verify the main functions which are to manage, monitor, and upgrade the communication system network elements
- ❑ **Management Information Base:** The MIB tests are designed to verify (i) standard MIBs including MIB II and (ii) enterprise MIBs specific to the system.
- ❑ **Graphical User Interface:** The GUI tests are designed to verify the interface to the users of an application. These tests verify different components (objects) such as icons, menu bars, dialogue boxes, scroll bars, list boxes, and radio buttons.
- ❑ **Security:** Security tests are designed to verify that the system meets the security requirements: *confidentiality*, *integrity*, and *availability*.
- ❑ **Feature:** Feature tests are designed to verify any additional functionalities which are defined in the requirement specifications but not covered in the above categories.

Communication systems

Module

Logging and tracing

Element management
systems

Management information
base

Graphical user interface

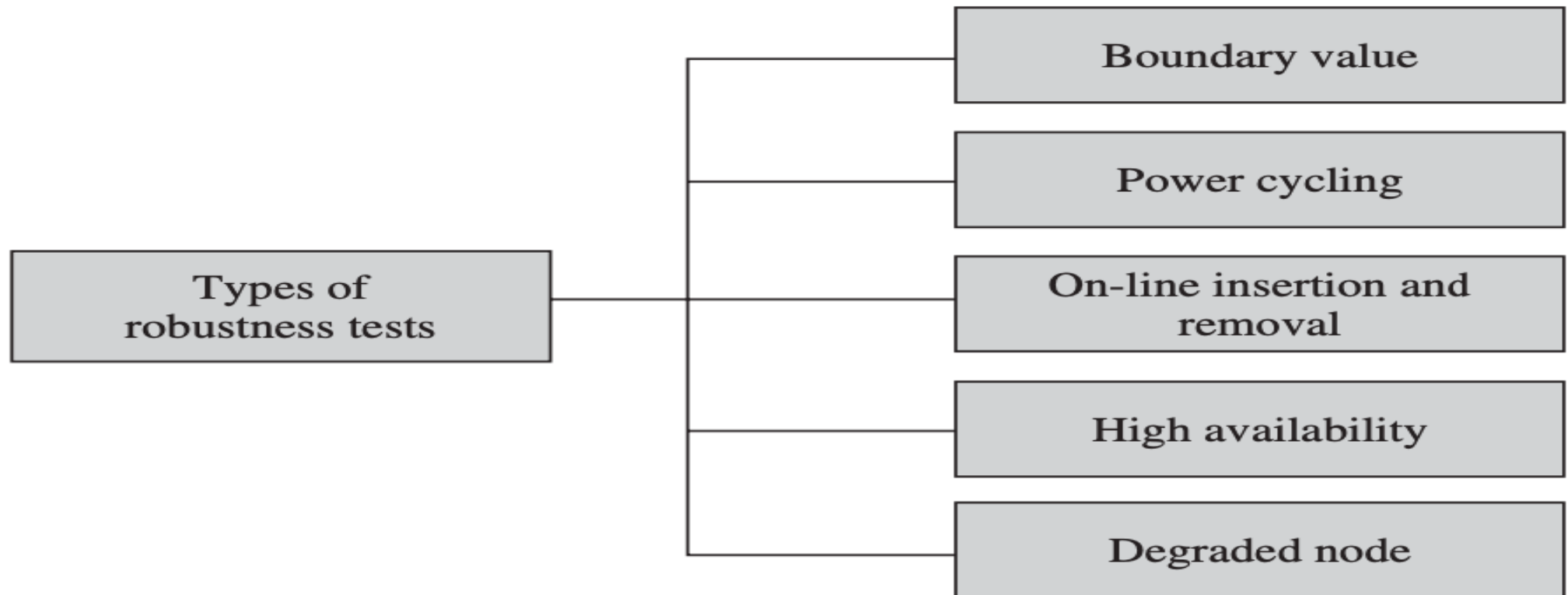
Security

Feature



ROBUSTNESS TESTS

❑ **Robustness means** how sensitive a system is to **erroneous input** and changes in its operational environment.



ROBUSTNESS TESTS

- ❑ **Boundary value tests** are designed to cover boundary conditions, special values, and system defaults.
- ❑ **Power cycling tests** are executed to ensure that, when there is a power glitch in a deployment environment, the system can recover from the glitch to be back in normal operation after power is restored.
- ❑ **On-line insertion and removal (OIR)** tests are designed to ensure that online insertion and removal of modules, incurred during both idle and heavy load operations, are gracefully handled and recovered.
- ❑ **High-availability tests** are designed to verify the redundancy of individual modules, including the software that controls these modules. The goal is to verify that the system gracefully and quickly recovers from hardware and software failures without adversely impacting the operation of the system. The concept of high availability is also known as **fault tolerance**.
- ❑ **Degraded node (also known as failure containment) tests** verify the operation of a system after a portion of the system becomes non operational. It is a useful test for all mission-critical applications.

Boundary value

Power cycling

On-line insertion and
removal

High availability

Degraded node



2.5. System Testing-Assignment

1. What are the differences between
 - i. Configuration, compatibility, and interoperability testing?
 - ii. Performance, stress, and scalability testing?
 - iii. load testing and stress testing?
 - iii. performance and speed?
 - iv. safety and reliability testing and security testing?
1. Discuss the importance of regression testing when developing a new software release. What test cases from the test suite would be more useful in performing a regression test?
2. What is the similarity between software safety and fault tolerance?
3. For each of the following situations, explain whether it is a hazard or a mishap:
 - (a) Water in a swimming pool becomes electrified.
 - (b) A room fills with carbon dioxide.
 - (c) A car stops abruptly.
 - (d) A long-distance telephone company suffers an outage.
 - (e) A nuclear weapon is destroyed in an unplanned manner



2.6. Acceptance Testing

- **Acceptance Testing:** A product is ready to be delivered to the customer after the system test group is satisfied with the product by performing system-level tests.
- **Customers execute** acceptance tests based on their expectations from the product.
- The services offered by a software product may be used by **millions of users**.
- Acceptance testing is a formal testing conducted to determine whether a system satisfies its **acceptance criteria**—**the criteria the system must satisfy to be accepted by the customer**.
- It helps the customer to determine whether or not to accept the system.
- **Types of Acceptance Testing:** There are two categories of acceptance testing:
 - **User acceptance testing:** The UAT is conducted by the **customer** to ensure that system satisfies the contractual acceptance criteria before being signed off as meeting user needs.
 - **Business acceptance testing:** The BAT is undertaken within the **development organization**



2.6. Acceptance Criteria

- **Acceptance Criteria:** The acceptance criteria **must be defined and agreed upon** between the **supplier** and the **customer** to avoid any kind of protracted arguments.
- The acceptance criteria document is a part of the contract **in the case of an outsourced development** under the OEM(original equipment manufacturer) agreement.
- In general, the marketing organization of the **buyer defines the acceptance criteria**.
- **It is useful to focus on the following three major objectives of acceptance testing for pragmatic reasons:**
 - Confirm that the system meets the **agreed-upon criteria**.
 - Identify and resolve **discrepancies**, if there are any. The sources of discrepancies and mechanisms for resolving them
 - Determine the readiness of the system for cut-over to live operations. The final acceptance of a system for deployment is conditioned upon the outcome of the acceptance testing. The acceptance test team produces an acceptance test report which outlines the **acceptance conditions**.



2.6. Acceptance Criteria

- **Acceptance Criteria**: Acceptance criteria are defined on the basis of these multiple facets of quality attributes. These attributes determine the presence or absence of **quality in a system**.
- **Buyers, or Customers**, should think through the relevance and relative importance of these attributes in their unique situation at the time of formulating the acceptance criteria.

Acceptance Criteria Quality Attributes:

- **Functional Correctness and Completeness**
- **Accuracy**
- **Data Integrity**
- **Backup and Recovery**
- **Competitive Edge**
- **Usability**
- **Performance**
- **Start-Up Time**
- **Stress**
- **Reliability and Availability**
- **Maintainability and Serviceability**
- **Robustness**
- **Timeliness**
- **Confidentiality and Availability**
- **Compatibility and Interoperability**
- **Compliance**
- **Installability and Upgradability**
- **Scalability**
- **Documentation**



2.6. Selection of Acceptance Criteria

- **Selection of Acceptance Criteria:** The customer needs to select **a subset of the quality attributes** and **prioritize them** to suit their specific situation. Next, the customer identifies the acceptance criteria for each of the selected quality attributes. When the customer and the software vendor reach an agreement on the acceptance criteria, both parties must keep in mind that satisfaction of the **acceptance criteria is a trade-off between time, cost, and quality.**
- Only business goals and priorities can determine the degree of “less than perfect” that is acceptable to both the parties. Ultimately, the acceptance criteria must be **related to the business goals of the customer’s organization.**

Eg: Usability and maintainability take precedence over performance and reliability for a word processor software. On the other hand, it might be the other way around for a real-time operating system or telecommunication software.



Acceptance Test Plan

- **Acceptance Test Plan :** Planning for acceptance testing begins as soon as the acceptance criteria are known. Early development of an acceptance test plan (ATP) gives us a good picture of the final product.
- The purpose of an ATP is to develop a detailed outline of the process to test the system prior to making a transition to the actual business use of the system.
- Often, the ATP is delivered by the vendor as a contractual agreement, so that the business acceptance testing can be undertaken within the vendor's development organization to ensure that the system eventually passes the acceptance test.
- In developing an ATP, emphasis is put on demonstrating that the system works according to the customer's expectation, rather than passing a set of comprehensive tests
- An ATP needs to be written and executed by the customer's special user group.



2.6. Acceptance Testing-Outline of ATP

1.Introduction

2.Acceptance test category. For each category of acceptance criteria:

- (a) Operational environment
- (b) Test case specification
 - (i) Test case ID number
 - (ii) Test title
 - (iii) Test objective
 - (iv) Test procedure

3.Schedule

4.Human resources



2.6. Acceptance Test Execution

- **Acceptance Test Execution** :The activity includes the following detailed actions:
- The developers train the customer **on the usage of the system**.
- **The developers and the customer coordinate** the fixing of any problem discovered during acceptance testing.
- **The developers and the customer** resolve the issues arising out of **any acceptance criteria discrepancy**.

ACC Document Information

1. ACC number	A unique number
2. Acceptance criteria affected	Existing acceptance criteria
3. Problem/issue description	Brief description of issue
4. Description of change required	Description of changes needed to be done to original acceptance criterion
5. Secondary technical impacts	Description of impact it will have on system
6. Customer impacts	Impact it will have on end user
7. Change recommended by	Name of acceptance test engineer(s)
8. Change approved by	Name of approver(s) from both parties



2.6. Acceptance Test Report

- **Acceptance Test Report :**
- **Structure of Acceptance Test Status Report:**

1. Date	Acceptance report date
2. Test case execution status	Number of test cases executed today Number of test cases passing Number of test cases failing
3. Defect identifier	Submitted defect number Brief description of issue
4. ACC number(s)	Acceptance criteria change document number(s), if any
5. Cumulative test execution status	Total number of test cases executed Total number of test cases passing Total number of test cases failing Total number of test cases not executed yet



2.6. Acceptance Test Report

▪ Acceptance Test Report :

Structure of Acceptance Test Summary Report

1. Report identifier
2. Summary
3. Variances
4. Summary of results
5. Evaluation
6. Recommendations
7. Summary of activities
8. Approval



QUESTIONS PLEASE 😊

