Unit 4

What is Testing?

- “Testing is the process of executing a program with the intent of finding errors.”
- Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected.
- If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.
- Some commonly used terms associated with testing are:

i. **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

ii. **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

iii. **Test suite:** This is the set of all test cases with which a given software product is to be tested.

Aim of testing

- The aim of the testing process is to identify all defects existing in a software product.
- It is not possible to guarantee that the software is error free.
- This is because the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume.
- Even with this practical limitation of the testing process, the importance of testing should not be underestimated.
- Testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Software Testing Principles

- Software testing is a procedure of implementing software or the application to identify the defects or bugs.
- For testing an application or software, we need to follow some principles to make our product defects free, and that also helps the test engineers to test the software with their effort and time.
- Let us see the seven different testing principles, one by one:

1. Testing shows the presence of defects
2. Exhaustive Testing is not possible
3. Early Testing
4. Defect Clustering
5. Pesticide Paradox
6. Testing is context-dependent
7. Absence of errors fallacy

Software Testing Principles: 1. Testing shows the presence of defects

- The test engineer will test the application to make sure that the application is bug or defects free.
- While doing testing, we can only identify that the application or software has any errors.
- The primary purpose of doing testing is to identify the numbers of unknown bugs with the help of various methods and testing techniques because the entire test should be traceable to the customer requirement, which means that to find any defects that might cause the product failure to meet the client's needs.
- By doing testing on any application, we can decrease the number of bugs, which does not mean that the application is defect-free because sometimes the software seems to be bug-free while performing multiple types of testing on it.
- But at the time of deployment in the production server, if the end-user encounters those bugs which are not found in the testing process.

Software Testing Principles: Exhaustive Testing is not possible

- Sometimes it seems to be very hard to test all the modules and their features with effective and non- effective combinations of the inputs data throughout the actual testing process.
- Hence, instead of performing the exhaustive testing as it takes boundless determinations and most of the hard work is unsuccessful.
- So we can complete this type of variations according to the importance of the modules because the product timelines will not permit us to perform such type of testing scenarios.

Software Testing Principles: Early Testing

- Here early testing means that all the testing activities should start in the early stages of the software development life cycle's **requirement analysis stage** to identify the defects because if we find the bugs at an early stage, it will be fixed in the initial stage itself, which may cost us very less as compared to those which are identified in the future phase of the testing process.
- To perform testing, we will require the requirement specification documents; therefore, if the requirements are defined incorrectly, then it can be fixed directly rather than fixing them in another stage, which could be the development phase.

Software Testing Principles: Defect clustering

- The defect clustering defined that throughout the testing process, we can detect the numbers of bugs which are correlated to a small number of modules.
- We have various reasons for this, such as the modules could be complicated; the coding part may be complex, and so on.
- These types of software or the application will follow the **Pareto Principle**, which states that we can identify that approx.
- Eighty percent of the complication is present in 20 percent of the modules.
- With the help of this, we can find the uncertain modules, but this method has its difficulties if the same tests are performing regularly, hence the same test will not able to identify the new defects.

Software Testing Principles: Pesticide paradox

- This principle defined that if we are executing the same set of test cases again and again over a particular time, then these kinds of the test will not be able to find the new bugs in the software or the application.
- To get over these pesticide paradoxes, it is very significant to review all the test cases frequently.
- And the new and different tests are necessary to be written for the implementation of multiple parts of the application or the software, which helps us to find more bugs.

Software Testing Principles: Testing is context-dependent

- Testing is a context-dependent principle states that we have multiple fields such as e-commerce websites, commercial websites, and so on are available in the market.
- There is a definite way to test the commercial site as well as the e-commerce websites because every application has its own needs, features, and functionality.
- To check this type of application, we will take the help of various kinds of testing, different technique, approaches, and multiple methods. Therefore, the testing depends on the context of the application.

Software Testing Principles: Absence of errors fallacy

- Once the application is completely tested and there are no bugs identified before the release, so we can say that the application is 99 percent bug-free.
- But there is the chance when the application is tested beside the incorrect requirements, identified the flaws, and fixed them on a given period would not help as testing is done on the wrong specification, which does not apply to the client's requirements.
- The absence of error fallacy means identifying and fixing the bugs would not help if the application is impractical and not able to accomplish the client's requirements and needs.

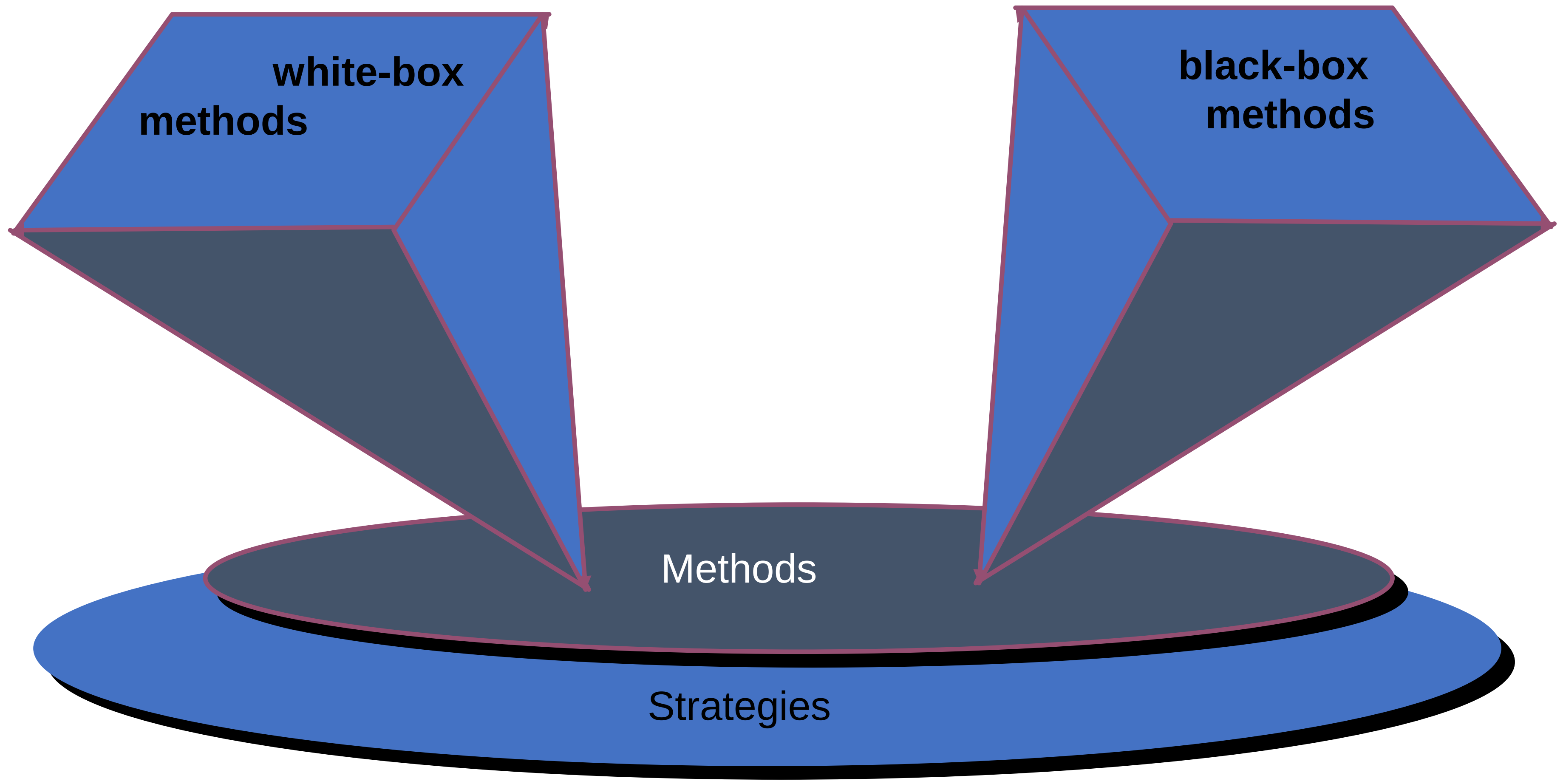
Software Testing

- **Software testing can be divided into two steps:**
  1. **Verification:** it refers to the set of tasks that ensure that software correctly implements a specific function.

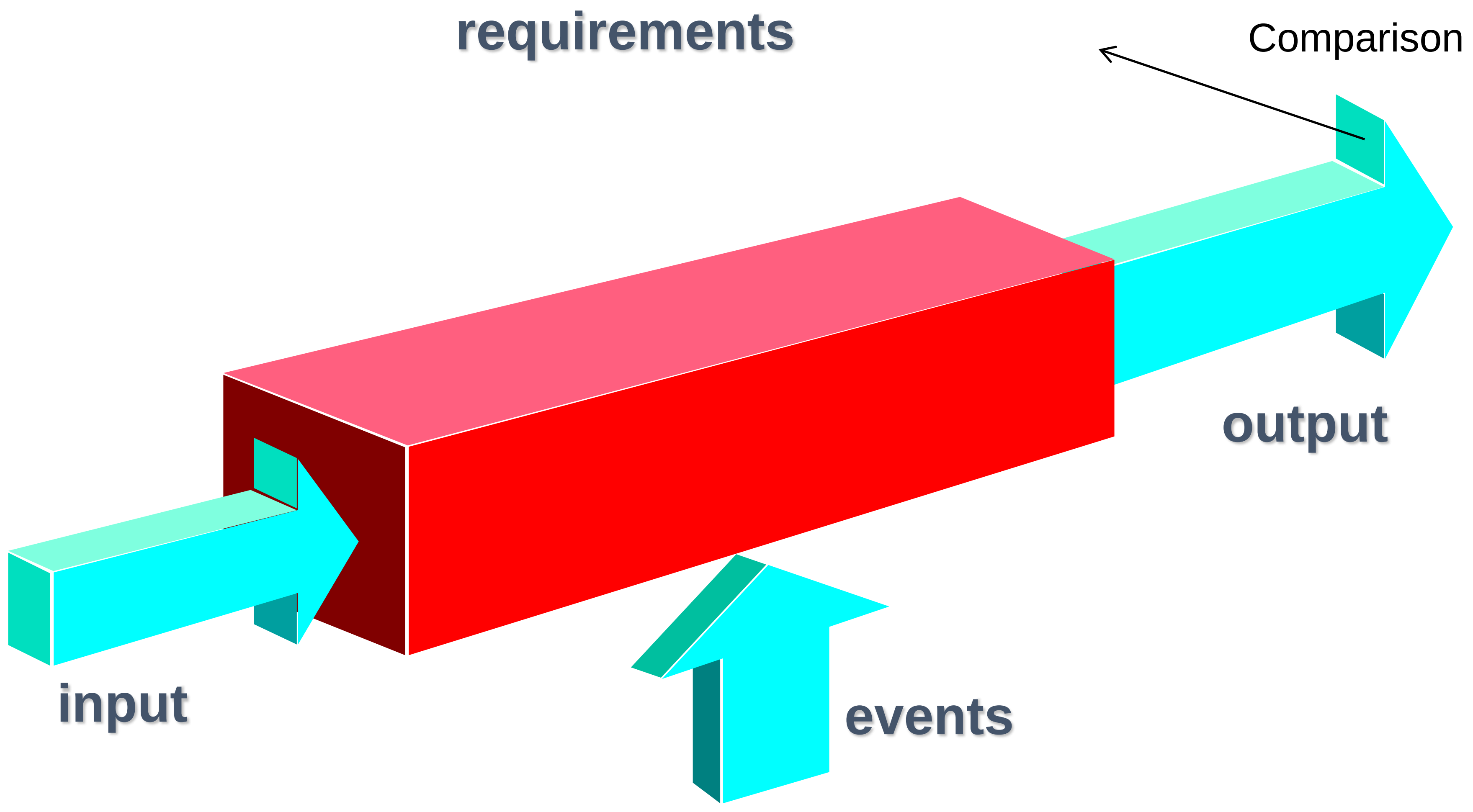
  2. **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

Methods of Software Testing



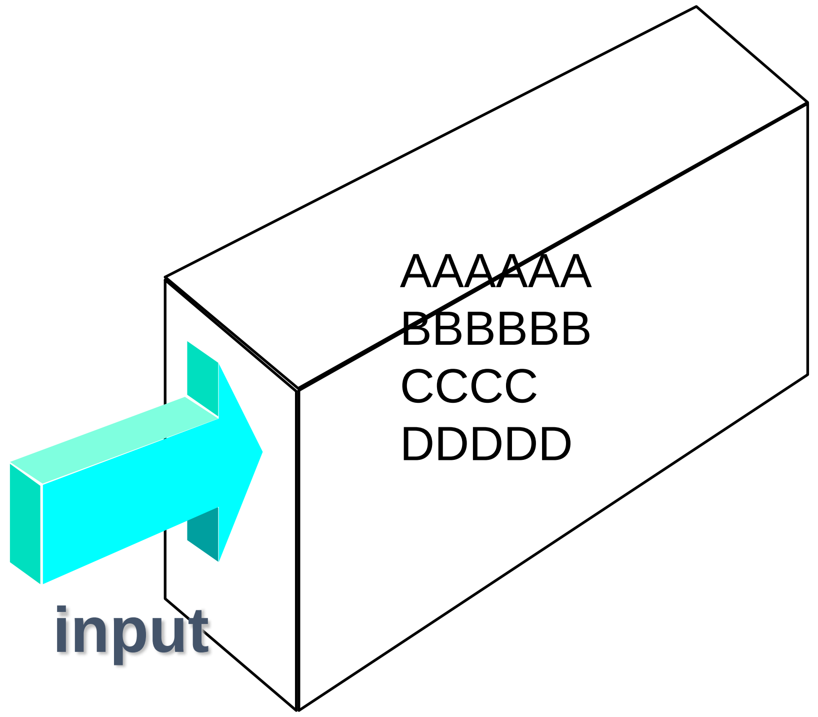
Black-Box Testing



Black-Box Testing

- Black box design treats the system as a black-box.
- It is also known as Functional Testing, Behavioral Testing, Data-Driven Testing, Input/Output driven Testing.
- Give input to the system, get the output, compare the output with the specification, if it matches, then the system is fine, otherwise error is there.
- Black-Box testing attempts to uncover the following
  - ✓Incorrect functions
  - ✓Missing functions
  - ✓Performance errors
  - ✓Initialization & Termination errors
  - ✓External Database access errors.

White-Box Testing



White-Box Testing

- Also known as Structural Testing, Clear-Box Testing, Open-Box Testing, Logic-Driven Testing.
- Using White-Box testing methods, the software engineer can derive test cases that.
  - ✓Guarantee that all independent paths within a module have been exercised at least once.
  - ✓Exercise all logical decisions on their true and false sides.
  - ✓Execute all loops at their boundaries and within their operational bounds.
  - ✓Exercise internal data structures to ensure their validity.

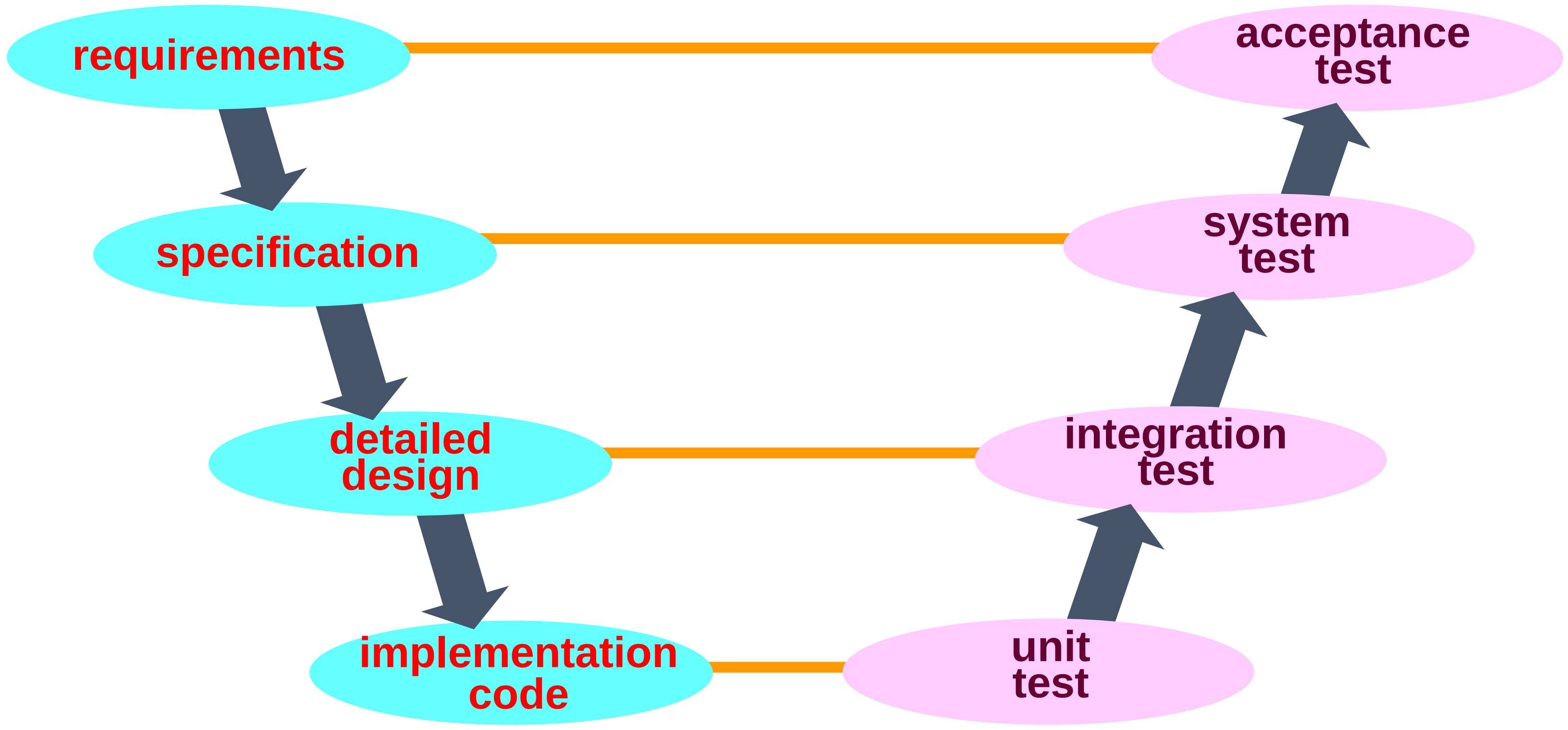
- **white-box testing**: the internal structure of the software is taken into account to derive the test cases

Difference Between Black Box Testing and White Box Testing

BLACK BOX TESTING	WHITE BOX TESTING
Internal workings of an application are not required.	Knowledge of the internal workings is must.
Also known as closed box/data driven testing.	Also knwon as clear box/structural testing.
End users, testers and developers.	Normally done by testers and developers.
THis can only be done by trial and error method.	Data domains and internal boundaries can be better tested.

Test process in software development

V-model:



Unit Testing

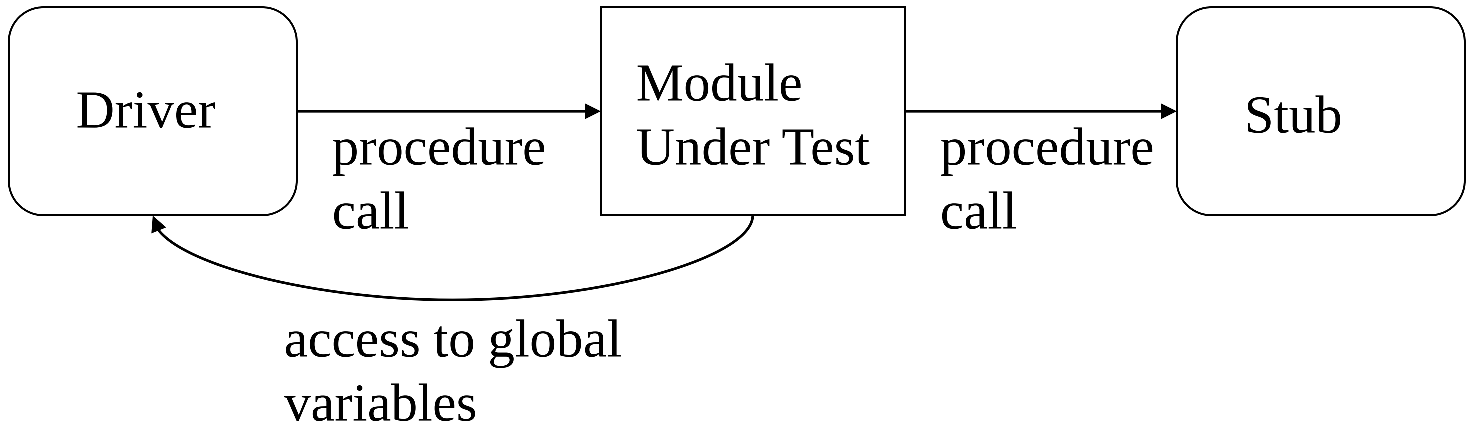
- Involves testing a single isolated module

- Note that unit testing allows us to isolate the errors to a single module
    - we know that if we find an error during unit testing it is in the module we are testing

- Modules in a program are not isolated, they interact with each other. Possible interactions:
    - calling procedures in other modules
    - receiving procedure calls from other modules
    - sharing variables

- For unit testing we need to isolate the module we want to test, we do this using two things
    - drivers and stubs

Drivers and Stubs



- Driver and Stub should have the same interface as the modules they replace
- Driver and Stub should be simpler than the modules they replace

Drivers and Stubs

- **Driver:** A program that calls the interface procedures of the module being tested and reports the results
  - A driver simulates a module that calls the module currently being tested
- **Stub:** A program that has the same interface as a module that is being used by the module being tested, but is simpler.
  - A stub simulates a module called by the module currently being tested

Integration Testing

- The modules that may work properly and independently may not work when they are integrated.

- Integration Testing will test whether the modules work well together.

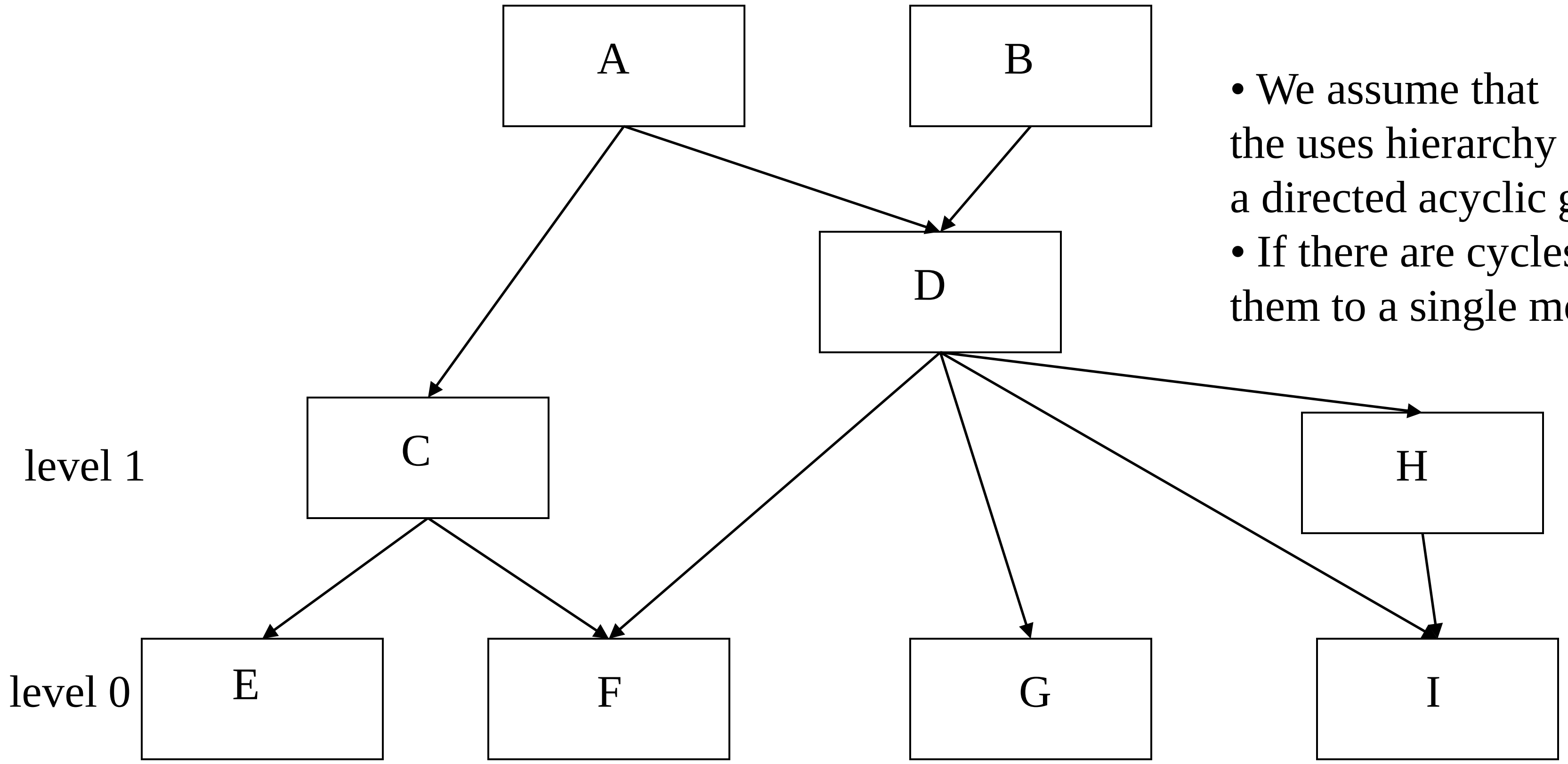
Integration Testing

- This will check whether the design is correct.
- Integration testing: Integrated collection of modules tested as a group or partial system
- Integration plan specifies the order in which to combine modules into partial systems

Integration Testing

- Different approaches to integration testing
  - Bottom-up
  - Top-down
  - Big-bang
  - Sandwich

Module Structure



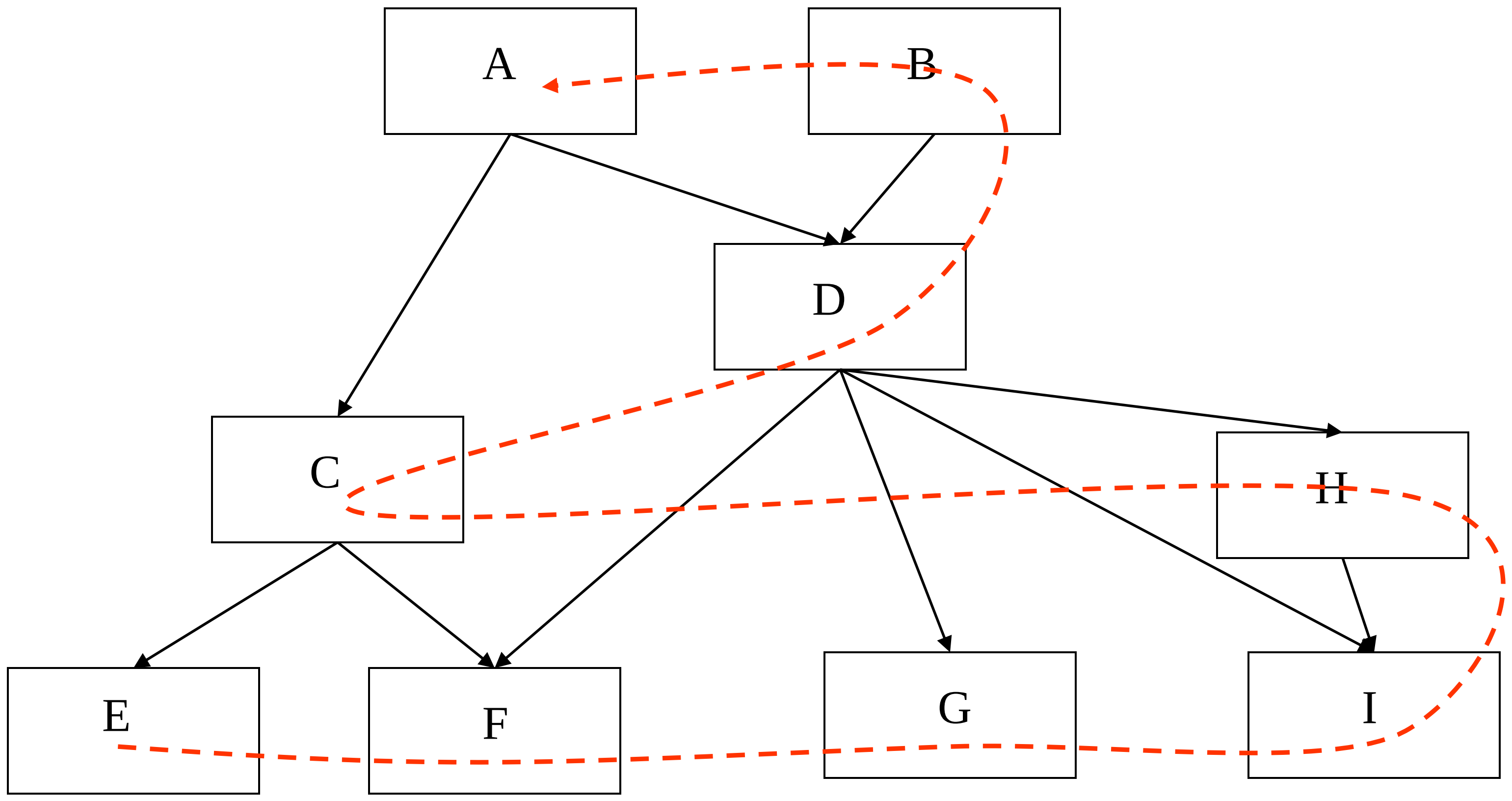
• We assume that the uses hierarchy is a directed acyclic graph.
• If there are cycles merge them to a single module

• A uses C and D; B uses D; C uses E and F; D uses F, G, H and I; H uses I
• Modules A and B are at level 3; Module D is at level 2
Modules C and H are at level 1; Modules E, F, G, I are at level 0
• level 0 components do not use any other components
• level *i* components use at least one component on level *i*-1 and no component at a level higher than *i*-1

Bottom-Up Integration

- Modules at lower levels are tested using the previously tested higher level modules.
- Non-terminal modules are not tested in isolation
- Requires a module driver for each module to feed the test case input to the interface of the module being tested
  - However, stubs are not needed since we are starting with the terminal modules and use already tested modules when testing modules in the lower levels

Bottom-up Integration



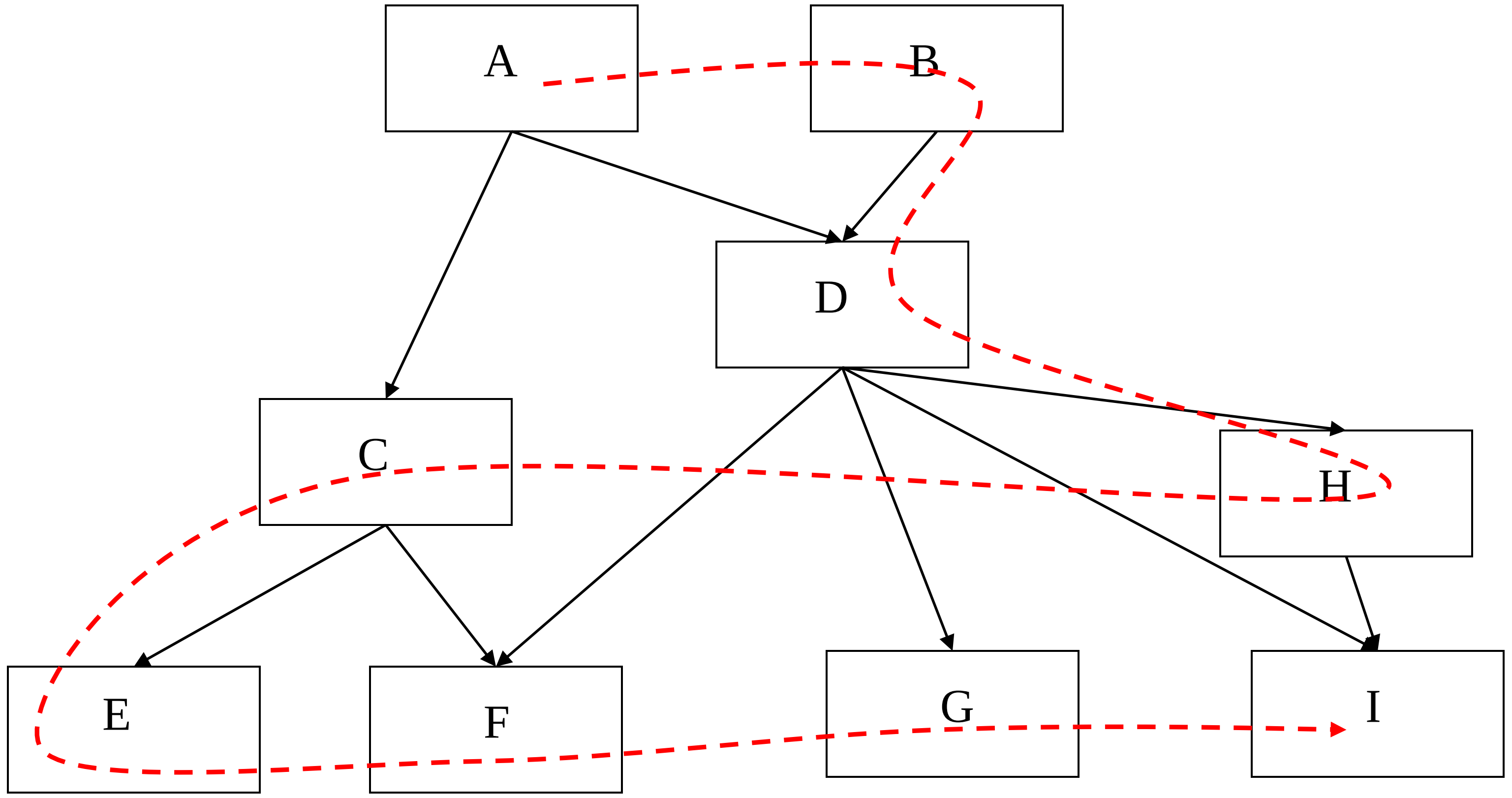
Top-down Integration

- Only modules tested in isolation are the modules which are at the highest level

- After a module is tested, the modules directly called by that module are merged with the already tested module and the combination is tested

- Requires stub modules to simulate the functions of the missing modules that may be called
  - However, drivers are not needed since we are starting with the modules which is not used by any other module and use already tested modules when testing modules in the higher levels

Top-down Integration



Other Approaches to Integration

- **Sandwich Integration**
  - Compromise between bottom-up and top-down testing
  - Simultaneously begin bottom-up and top-down testing and meet at a predetermined point in the middle
  - Instead of completely going for top down or bottom up, a layer is identified in between.
  - Above this layer we go for top down and below this layer bottom up.

- **Big Bang Integration**
  - Every module is unit tested in isolation
  - After all of the modules are tested they are all integrated together at once and tested
  - No driver or stub is needed
  - However, in this approach, it may be hard to isolate the bugs!

System Testing

- At the system testing level, the system is tested as a whole, and primarily functional testing techniques are used to test the system.

System Testing

- System testing ensures that each system function works as expected and it also tests for non-functional requirements like performance, security, reliability, stress, load, etc.

- This is the only phase of testing which tests both functional and non-functional requirements of the system.

System Testing

- A team of the testing persons does the system testing under the supervision of a test team leader.

- We also review all associated documents and manuals of the software.

System Testing

- This verification activity is equally important and may improve the quality of the final product.
- A proper impact analysis should be done before fixing the defect.
- Sometimes, if the system permits, instead of fixing the defects, they are just documented and mentioned as the known limitations.

System Testing

- This may happen in a situation when fixing is very time consuming or technically it is not possible in the present design, etc.

System Testing

- Progress of system testing also builds confidence in the development team as this is the first phase in which the complete product is tested with a specific focus on the customer’s expectations.

- After the completion of this phase, customers are invited to test the software.

System Testing Types

- Regression Testing
- Load Testing
- Smoke Testing
- Usability Testing
- Security Testing
- Performance Testing
- Stress Testing

System Testing: Regression Testing

- **REGRESSION TESTING** is a type of software testing that intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it. Rerunning of tests can be on both functional and non-functional tests.
- Reasons for Regression Testing
- The need for Regression Testing could arise due to any of the changes below:

i. Defect fix
ii. New feature
iii. Change in an existing feature
iv. Code refactoring
v. Change in technical design / architecture
vi. Change in configuration / environment (hardware, software, network)

Smoke Testing

- Smoke testing covers most of the major functions of the software but none of them in depth.
- The result of this test is used to decide whether to proceed with further testing.
- If the smoke test passes, go ahead with further testing. If it fails, halt further tests and ask for a new build with the required fixes.
- If an application is badly broken, detailed testing might be a waste of time and effort.

Smoke Testing

- Smoke test helps in exposing integration and major problems early in the cycle.

- It can be conducted on both newly created software and enhanced software.

- Smoke test is performed **manually or with the help of automation tools/scripts**. If builds are prepared frequently, it is best to automate smoke testing.

Usability Testing

- **USABILITY TESTING** is a type of software testing done from an end-user's perspective to determine if the system is easily usable. It falls under non-functional testing.

- Usability Testing uses the Black Box Testing method where internal structure of the system is unknown and is always **executed manually** because of the need for human interaction and perception.

Security Testing

- **SECURITY TESTING** is a type of software testing that intends to uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders.
- It is a type of non-functional testing.
- There are four main focus areas to be considered in security testing (Especially for web sites/applications):
- **Network security:** This involves looking for vulnerabilities in the network infrastructure (resources and policies).

Security Testing

- **System software security:** This involves assessing weaknesses in the various software (operating system, database system, and other software) the application depends on.
- **Client-side application security:** This deals with ensuring that the client (browser or any such tool) cannot be manipulated.
- **Server-side application security:** This involves making sure that the server code and its technologies are robust enough to fend off any intrusion.

Performance Testing

- **PERFORMANCE TESTING** is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load.
- It is a type of non-functional testing.
- Performance Testing mainly focuses on the following software quality attributes:
- **Responsiveness:** The ability of software to respond quickly or complete assigned tasks within a reasonable time.
- **Concurrency:** The ability of software to service multiple requests to the same resources at the same time.

Performance Testing

- **Reliability:** The ability of software to perform a required function under stated conditions for the stated period of time without any errors.
- **Stability:** The ability of software to remain stable under varying loads or over time.
- **Scalability:** The measure of software’s ability to increase or decrease in performance in response to changes in software’s processing demands.

LOAD Testing

- **Load testing** is used to identify whether the **infrastructure used for hosting** the application is **sufficient or not.**

- It is used to find if the performance of the application is sustainable when it is at the peak of its user load.

LOAD Testing

- It tells us **how many simultaneous users can the application handle** and the scale of the application required in terms of hardware, network capacity etc., so that more users could access the application

- It helps to **identify the maximum operating capacity** of an application as well as any **bottlenecks** and determine **which element is causing degradation**.
- E.g. If the number of users are increased then how much CPU, memory will be consumed, what is the network and bandwidth response time.

Stress Testing

- It is a type of **non-functional testing**
- It involves **testing beyond normal operational capacity**, often to a **breaking point**, in order to observe the results
- It is a form of **software testing** that is used to determine the stability of a given system
- It put greater emphasis on **robustness, availability and error handling** under a heavy load, rather than on what would be considered correct behavior under normal circumstances

Stress Testing

- The goals of such tests may be to ensure the software does not crash in conditions of insufficient computational resources (such as memory, disk space, network request etc)
- Stress testing is also called fatigue testing
- The main **purpose of stress testing**: Make sure that the system **fails and recovers easily**, this quality is also known as **recoverability**.

- Described below is a description of a small system, a Video shop system.
-
- The following information is maintained by the system
- Video details- video id, video title, number of copies
- Members details- member id, name, address, phone, overdue fines
- Rental details- video id, copy number, member id, return date
- The software system needs to provide the following functionality:
  - to add, change and delete videos from the system
  - to add, change and delete members' information
  - to make inquiries on videos using either video id or video title
  - to make inquiries on members
  - to provide a report of the videos in stock
  - to provide a report of the members overdue fines
-
  - **For this problem, do the following:**
  -

a) Identify all ILFs, EIFs, Data Element Type (DETs), EIs, EOs, and EQs and calculate their complexities.

Acceptance Testing

- Acceptance Testing is done by the customer to test whether the system is meeting the requirements
- Acceptance Testing is of two types
    - ✓ Alpha Testing
    - ✓ Beta Testing

Acceptance Testing

Alpha Testing –

The Alpha test is conducted in the developer's environment by the end-users.

The environment might be simulated, with the developer and the typical end-user present for the testing.

The end-user uses the software and records the errors and problems.

Alpha test is conducted in a controlled environment.

Acceptance Testing

Beta Testing –

The Beta test is conducted in the end-user’s environment. The developer is not present for the beta testing.

The beta testing is always in the real-world environment which is not controlled by the developer.

The end-user records the problems and reports it back to the developer at intervals.

Based on the results of the beta testing the software is made ready for the final release to the intended customer base.

Black-Box Testing Techniques

Following are main techniques to black-box testing.

- Equivalence Class Partitioning
- Boundary Value Analysis
- Cause-Effect Graphs.

Equivalence Class Partitioning

- Equivalence partitioning (EP) is a specification-based or black-box technique.
- It can be applied at any level of testing and is often a good technique to use first.
- The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes – the two terms mean exactly the same thing.
- In equivalence-partitioning technique we need to test only one condition from each partition.
  - This is because we are assuming that all the conditions in one partition will be treated in the same way by the software.
  - If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others.
  - Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

Guidelines for Equivalence Class Partitioning

- If the range condition is given as an input, then one valid and two invalid equivalence classes are defined.
- If a specific value is given as input, then one valid and two invalid equivalence classes are defined.
- If a member of set is given as an input, then one valid and one invalid equivalence class is defined.
- If Boolean no. is given as an input condition, then one valid and one invalid equivalence class is defined.

Equivalence Class Partitioning (Contd.)

- If testing is done for an input box accepting numbers from 1 to 1000 then there is no use in writing thousand test cases for all 1000 valid input numbers plus other test cases for invalid data.
- Using equivalence partitioning method above test cases can be divided into three sets of input data called as classes. Each test case is a representative of respective class.
- So in above example we can divide our test cases into three equivalence classes of some valid and invalid inputs.

Equivalence Class Partitioning (Contd.)

- **Test cases for input box accepting numbers between 1 and 1000 using Equivalence Partitioning:**

1) One input data class with all valid inputs. **Pick a single value from range 1 to 1000 as a valid test case**. If you select other values between 1 and 1000 then result is going to be same. So one test case for valid input data should be sufficient.

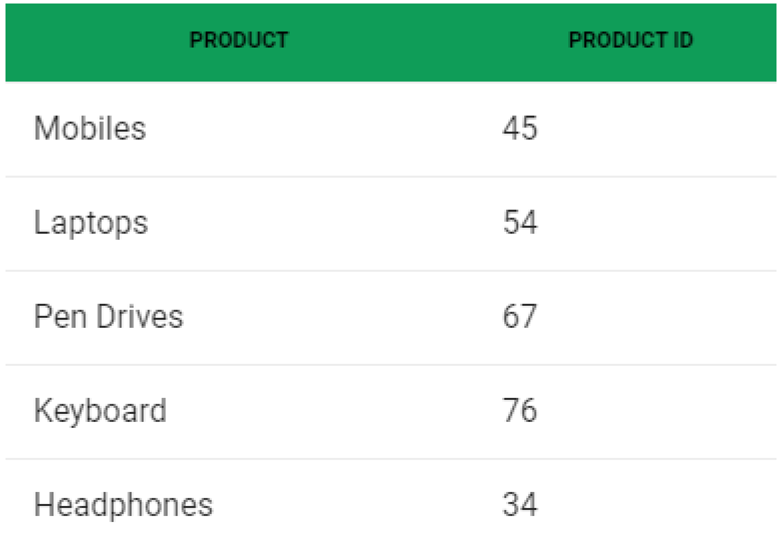
2) Input data class with all values **below lower limit**. I.e. **any value below 1**, as a invalid input data test case.

3) Input data with any value **greater than 1000** to represent third invalid input class.

- So using **equivalence partitioning** you have categorized all possible test cases into three classes. Test cases with other values from any class should give you the same result.

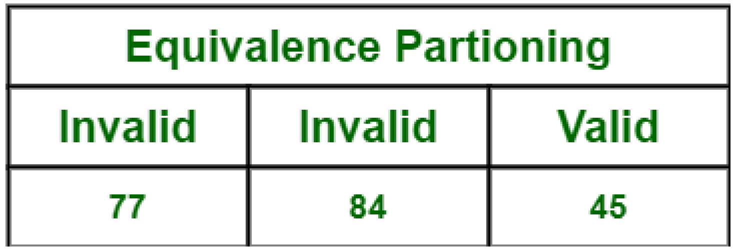
Equivalence Class Partitioning (Contd.)

- Let us consider an example of an online shopping site. In this site, each of products has specific product ID and product name. We can search for product either by using name of product or by product ID. Here, we consider search field that accepts only valid product ID or product name.
- Let us consider set of products with product IDs and user want to search for Mobiles. Below is table of some products with their product Id.



PRODUCT	PRODUCT ID
Mobiles	45
Laptops	54
Pen Drives	67
Keyboard	76
Headphones	34

- If the product ID entered by user is invalid then application will redirect customer or user to error page. If product ID entered by user is valid i.e. 45 for mobile, then equivalence partitioning method will show a valid product ID.



Equivalence Partioning		
Invalid	Invalid	Valid
77	84	45

Boundary Value Analysis

- It’s widely recognized that input values at the extreme ends of input domain cause more errors in system.
- More application **errors occur at the boundaries** of input domain.
- ‘Boundary value analysis’ testing technique is used to identify errors at boundaries rather than finding those exist in center of input domain.
- Boundary value analysis is a next part of Equivalence partitioning for designing test cases where test cases are selected at the edges of the equivalence classes.

Boundary Value Analysis (Contd.)

- **Test cases for input box accepting numbers between 1 and 1000 using Boundary value analysis:**

1) Test cases with test data exactly as the input boundaries of input domain i.e. values 1 and 1000 in our case.

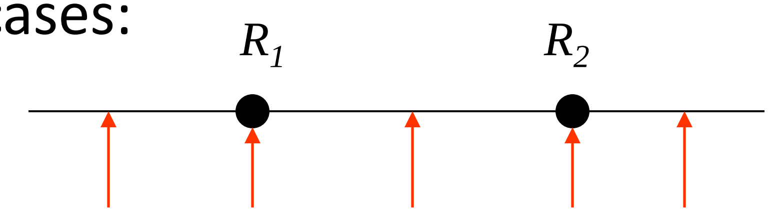
2) Test data with values just below the extreme edges of input domains i.e. values 0 and 999.

3) Test data with values just above the extreme edges of input domain i.e. values 2 and 1001.

- Boundary value analysis is often called as a part of stress and negative testing.

Boundary Value Analysis

- For each range [R_1, R_2] listed in either the input or output specifications, choose five cases:
  - Values less than R_1
  - Values equal to R_1
  - Values greater than R_1 but less than R_2
  - Values equal to R_2
  - Values greater than R_2



Cause-Effect Graphs

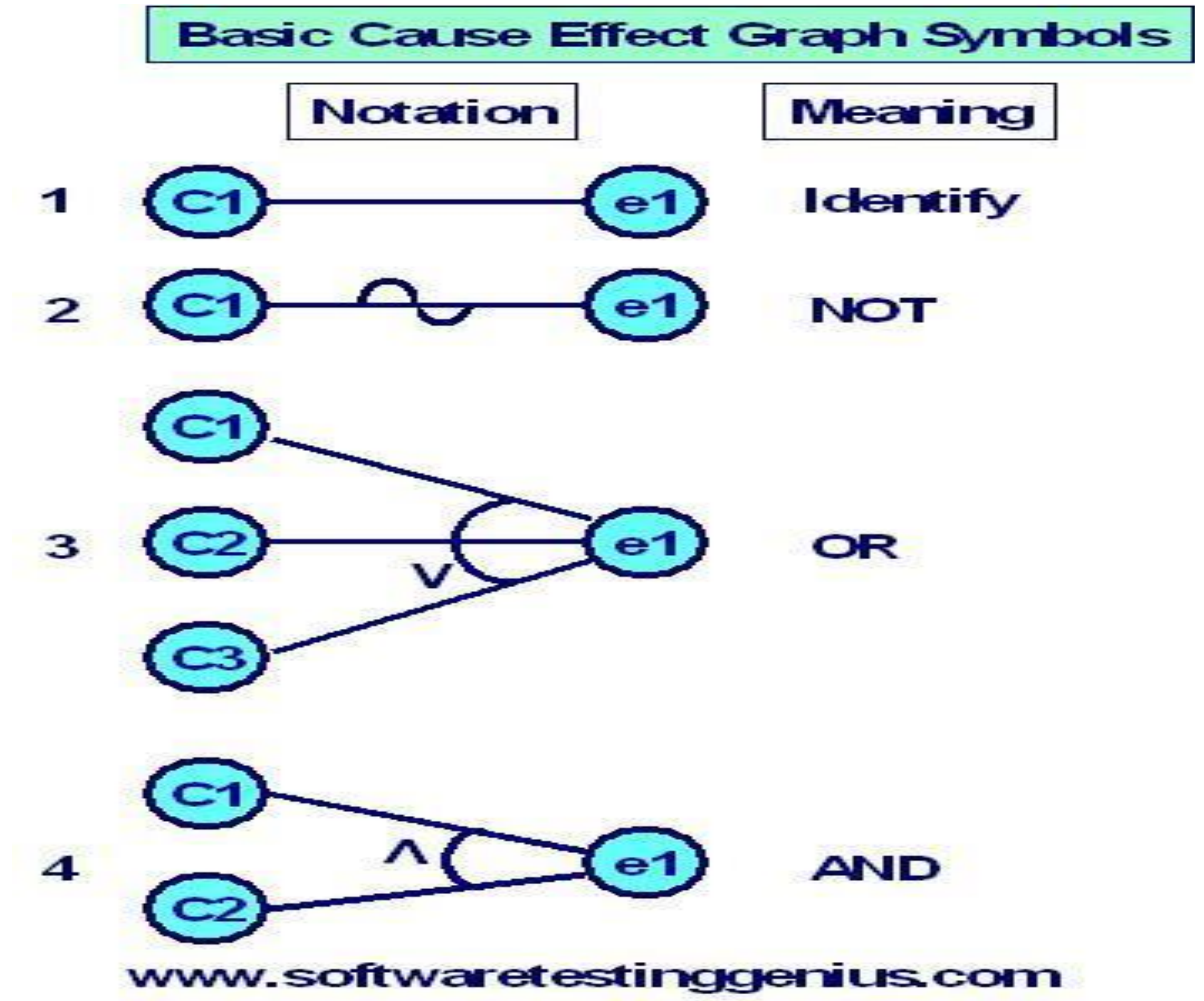
- This is basically a hardware testing technique adapted to software testing.
- It considers only the **desired external behaviour of a system.**
- This is a testing technique that **aids in selecting test cases that logically relate** Causes (inputs) to Effects (outputs) to produce test cases.
- A "Cause" represents a distinct input condition that brings about an internal change in the system.

Cause-Effect Graphs (Contd.)

- An **"Effect" represents an output condition, a system transformation or a state** resulting from a combination of causes.
- **According to Myer Cause & Effect Graphing is done through the following steps:**
- **Step – 1:** For a module, identify the input conditions (causes) and actions (effect).
- **Step – 2:** Develop a cause-effect graph.
- **Step – 3:** Transform cause-effect graph into a decision table.
- **Step – 4:** Convert decision table rules to test cases. Each column of the decision table represents a test case.

Cause-Effect Graphs (Contd.)

- **Basic symbols used in Cause-effect graphs are as under:**


Basic Cause Effect Graph Symbols
Notation
Meaning
1
C1
e1
Identify
2
C1
e1
NOT
C1
3
C2
V
e1
OR
C3
C1
4
Λ
e1
AND
C2
www.softwaretestinggenius.com

Cause-Effect Graphs (Contd.)

- Consider each node as having the value 0 or 1 where 0 represents the 'absent state' and 1 represents the 'present state'.
- Then the identity function states that if c1 is 1, e1 is 1 or we can say if c0 is 0, e0 is 0.
- The NOT function states that if C1 is 1, e1 is 0 and vice-versa. Similarly, OR function states that if C1 or C2 or C3 is 1, e1 is 1 else e1 is 0.
- The AND function states that if both C, and C2 are 1, e1 is 1; else e1 is 0.
- The AND and OR functions are allowed to have any number of inputs.

Cause-Effect Graphs (Contd.)

The character in column1 must be A or B. The character in column 2 must be a digit. If these 2 hold then file update is made. if the character in column1 is incorrect, message x is issued. If character in column2 is not a digit, message y is issued.

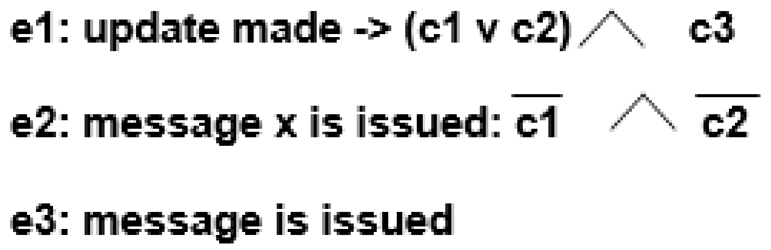
Causes are

c1: character in column1 is A

c2: character in column1 is B

c3: character in column2 is a digit

Effects are:

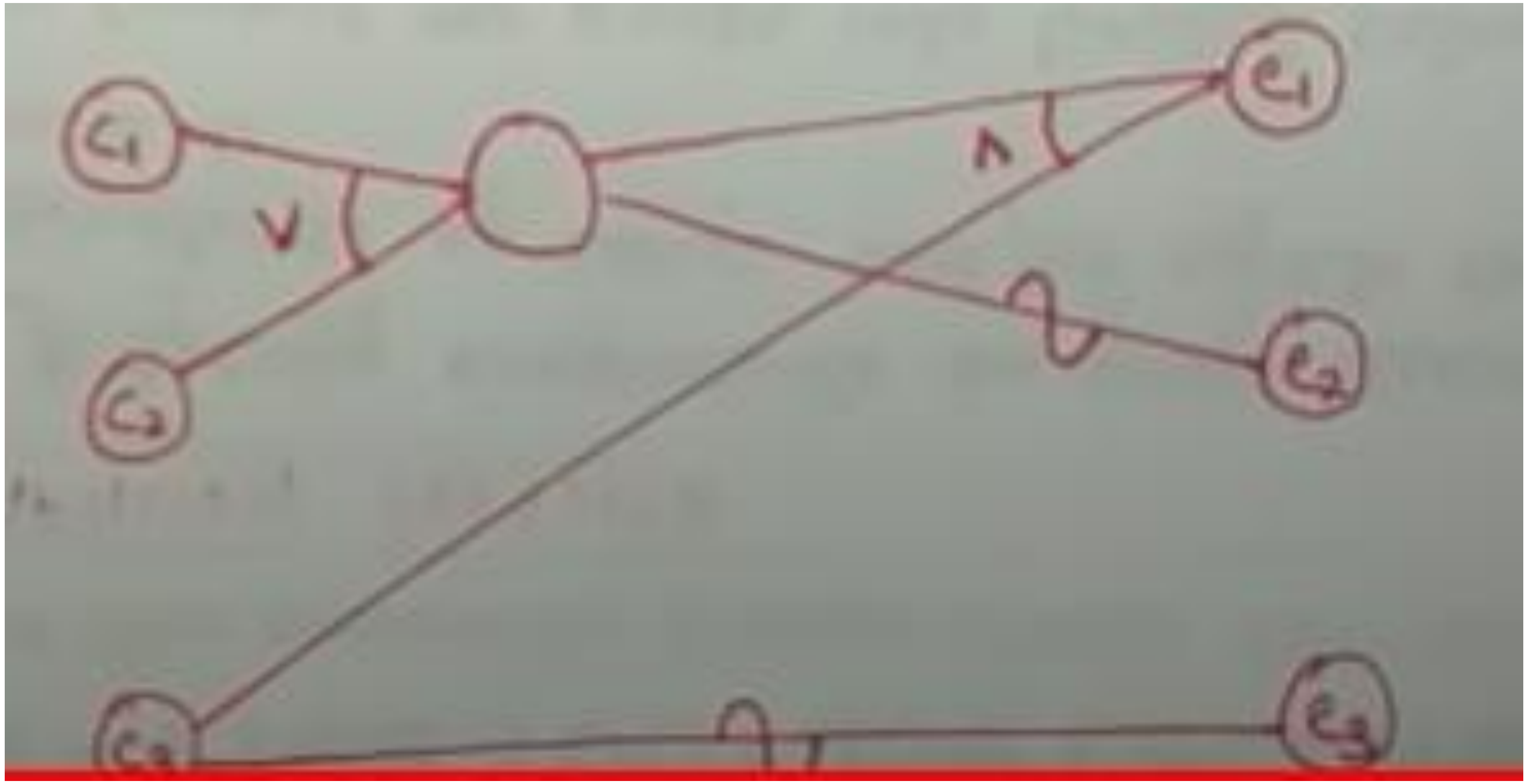


e1: update made -> $(c1 \vee c2) \wedge c3$

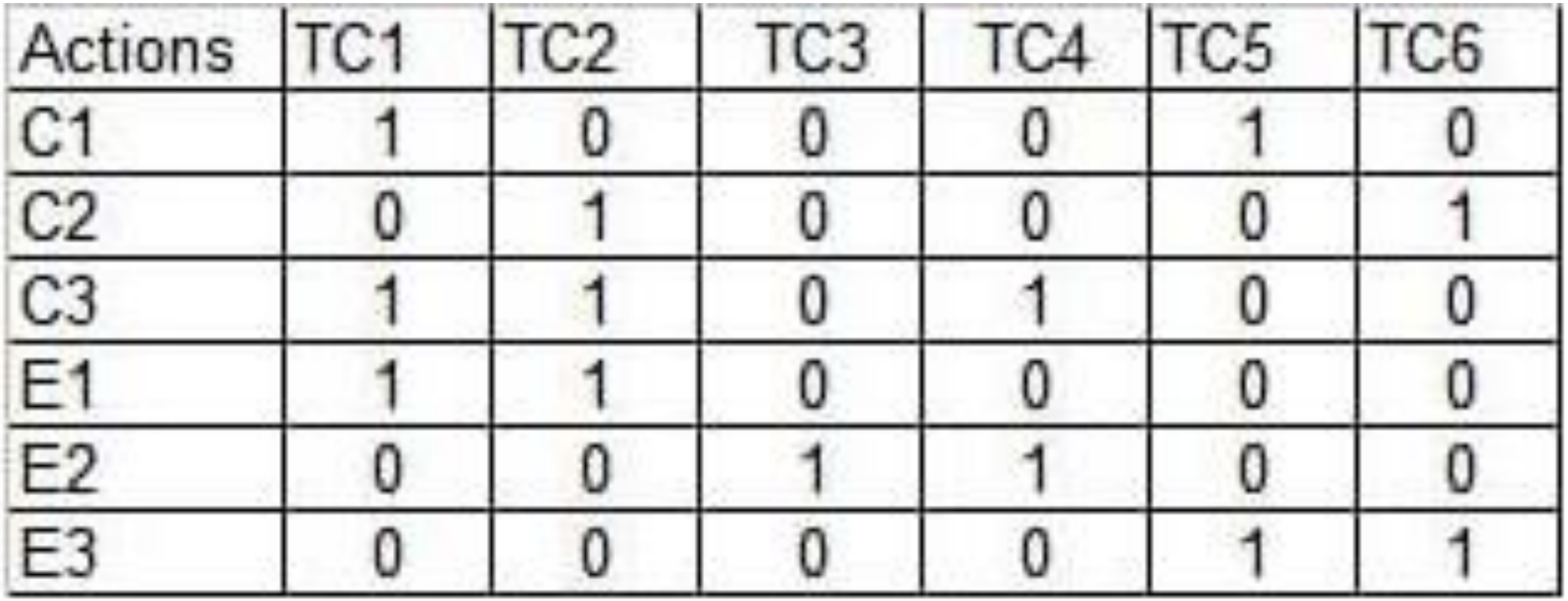
e2: message x is issued: $\overline{c1} \wedge \overline{c2}$

e3: message is issued

Cause-Effect Graphs (Contd.)

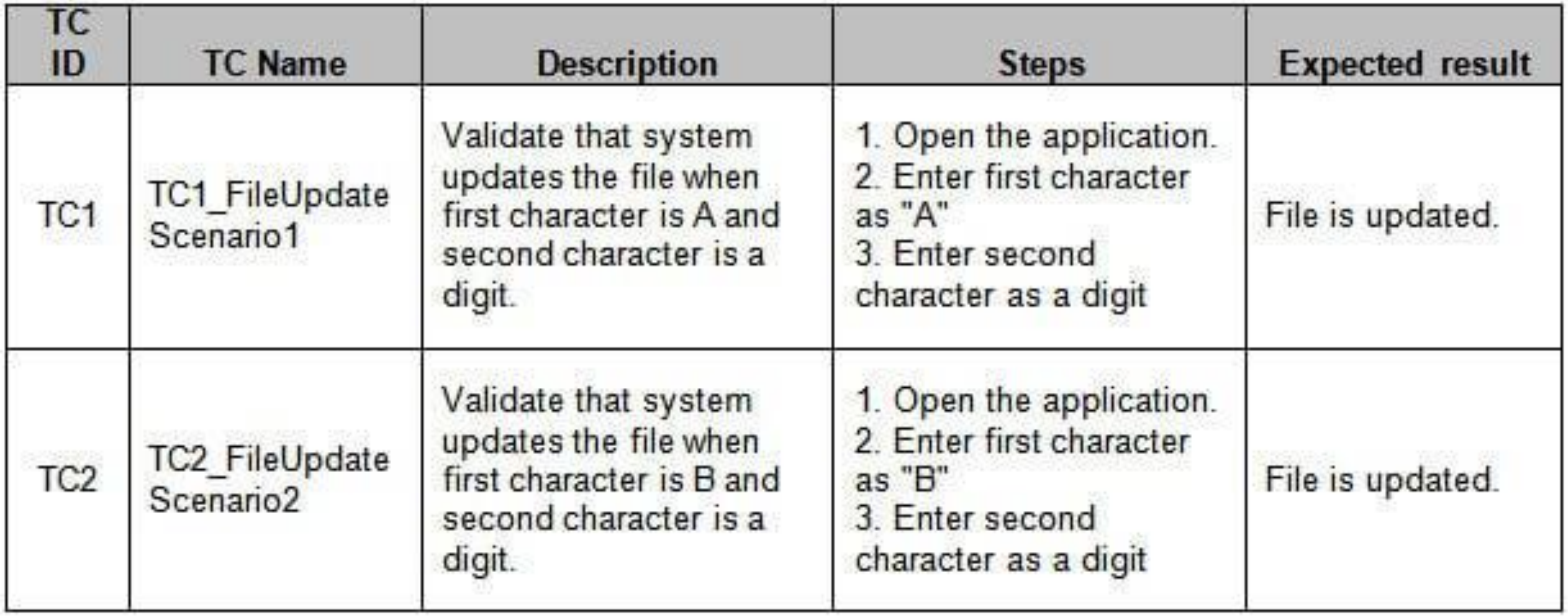


Cause-Effect Graphs (Contd.)



Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

Cause-Effect Graphs (Contd.)

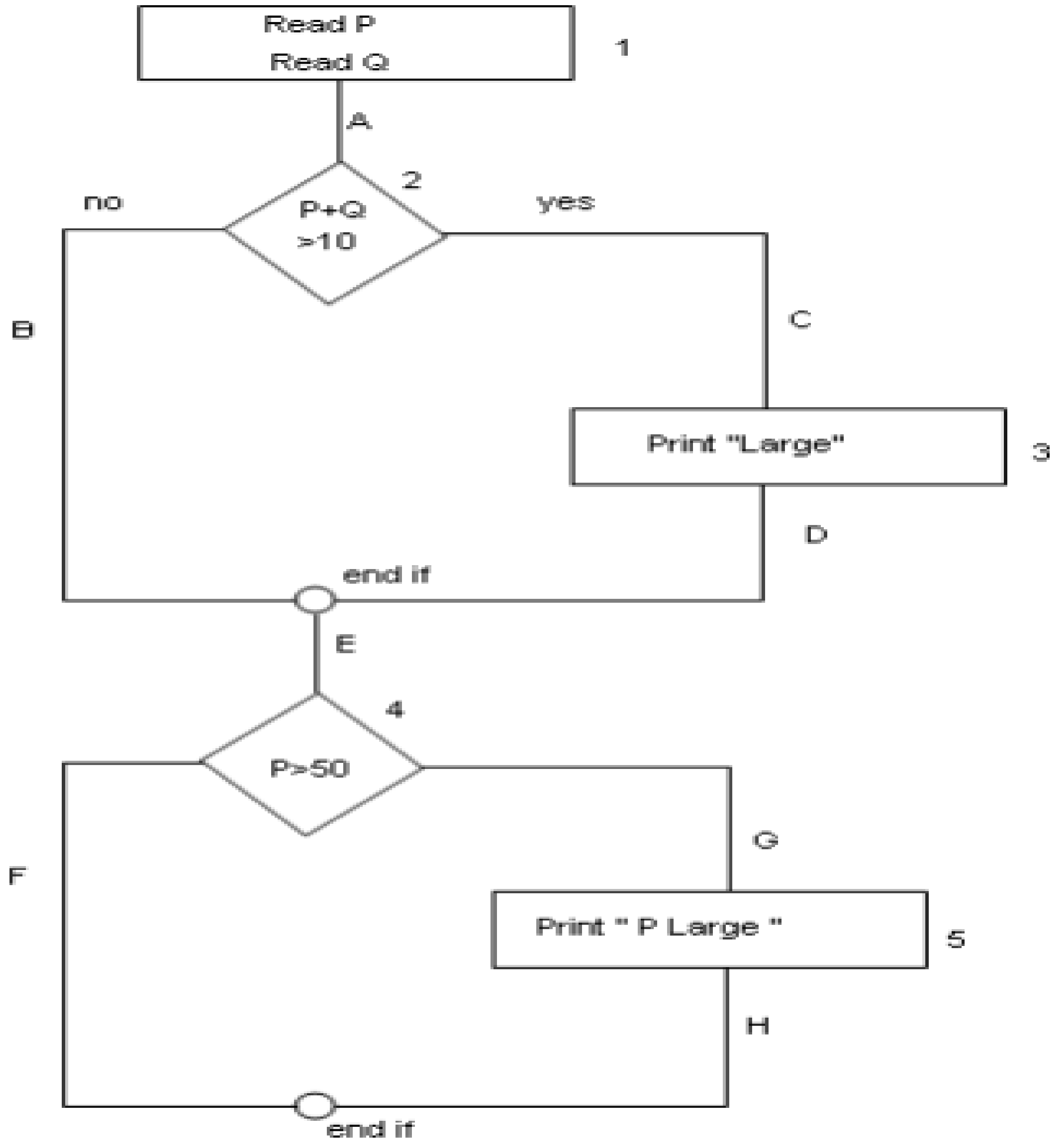


TC ID	TC Name	Description	Steps	Expected result
TC1	TC1_FileUpdate Scenario1	Validate that system updates the file when first character is A and second character is a digit.	1. Open the application. 2. Enter first character as "A" 3. Enter second character as a digit	File is updated.
TC2	TC2_FileUpdate Scenario2	Validate that system updates the file when first character is B and second character is a digit.	1. Open the application. 2. Enter first character as "B" 3. Enter second character as a digit	File is updated.

Coverage Metrics: White Box Testing Techniques

- Coverage metrics
  - ***Statement coverage***: all statements in the programs should be executed at least once
  - ***Branch coverage***: all branches in the program should be executed at least once
  - ***Path coverage***: all execution paths in the program should be executed at lest once
- The best case would be to execute all paths through the code,

```
Read P
Read Q
IF P+Q > 100 THEN
        Print “Large”
ENDIF
If P > 50 THEN
        Print “P Large”
ENDIF
```


Read P
Read Q
1
A
2
P+Q >10
no
yes
B
C
Print "Large"
3
D
end if
E
4
P>50
F
G
Print " P Large "
5
H
end if

Statement Coverage (SC):
To calculate Statement Coverage, find out the **shortest number of paths** following which all the nodes will be covered. Here by traversing through path 1A-2C-3D-E-4G-5H all the nodes are covered. So by traveling through only one path all the nodes 12345 are covered, so the Statement coverage in this case is 1.
Branch Coverage (BC):
To calculate Branch Coverage, find out the **minimum number of paths** which will ensure covering of all the edges. In this case there is no single path which will ensure coverage of all the edges at one go. By following paths 1A-2C-3D-E-4G-5H, maximum numbers of edges (A, C, D, E, G and H) are covered but edges B and F are left. To covers these edges we can follow 1A-2B-E-4F. By the combining the above two paths we can ensure of traveling through all the paths. Hence Branch Coverage is 2. The aim is to cover all possible true/false decisions.

Path Coverage (PC):
Path Coverage ensures covering of all the paths from start to end.
All possible paths are-
1A-2B-E-4F
1A-2B-E-4G-5H
1A-2C-3D-E-4G-5H
1A-2C-3D-E-4F
So path coverage is 4.
Thus for the above example SC=1, BC=2 and PC=4.

Statement Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
    print(“x is positive”);
  else
    print(“x is negative”);
  if (y >= 0)
    print(“y is positive”);
  else
    print(“y is negative”);
}
```

Following test set will give us statement coverage:
T_1 = {(x=12,y=5), (x=–1,y=35), (x=115,y=–13),(x=–91,y=–2)}

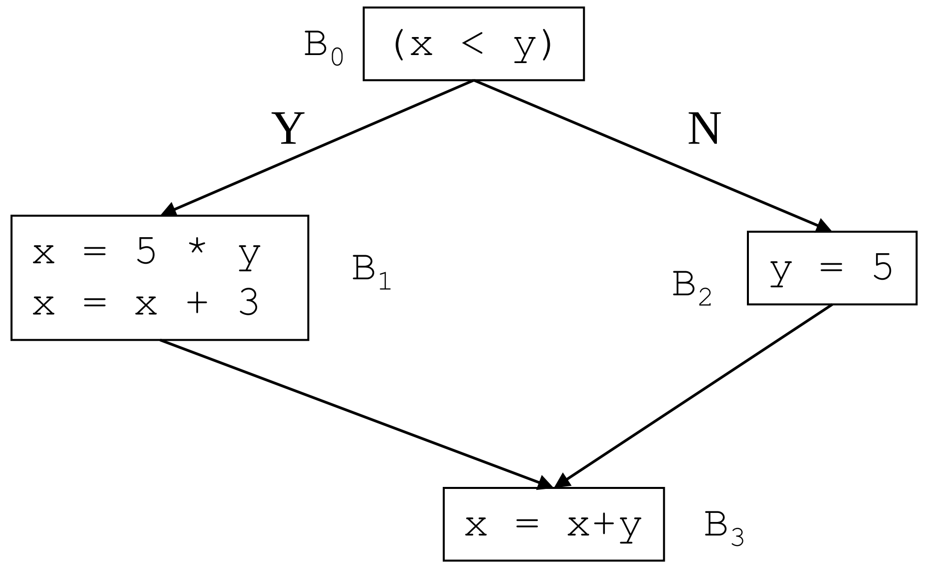
There are smaller test cases which will give us statement coverage too:
T_2 = {(x=12,y=–5), (x=–1,y=35)}

There is a difference between these two test sets though

Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
  - A ***basic block*** is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

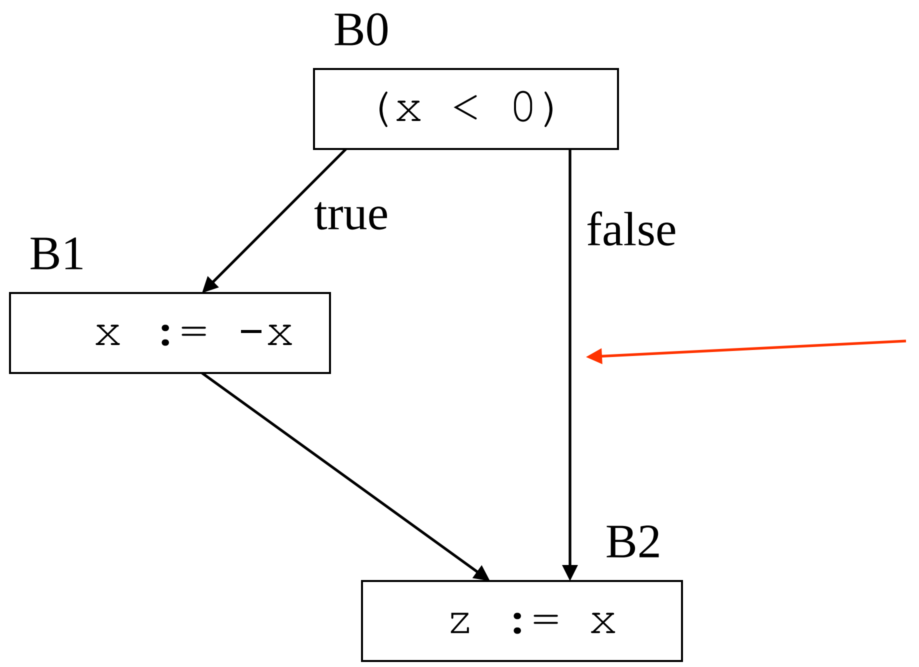
```
if (x < y) {
   x = 5 * y;
   x = x + 3;
}
else
   y = 5;
x = x+y;
```



- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

Branch Coverage

- Construct the control flow graph
- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, each edge of *P*'s control flow graph is traversed at least once



Test set {x=–1} does not execute this edge, hence, it does not give branch coverage

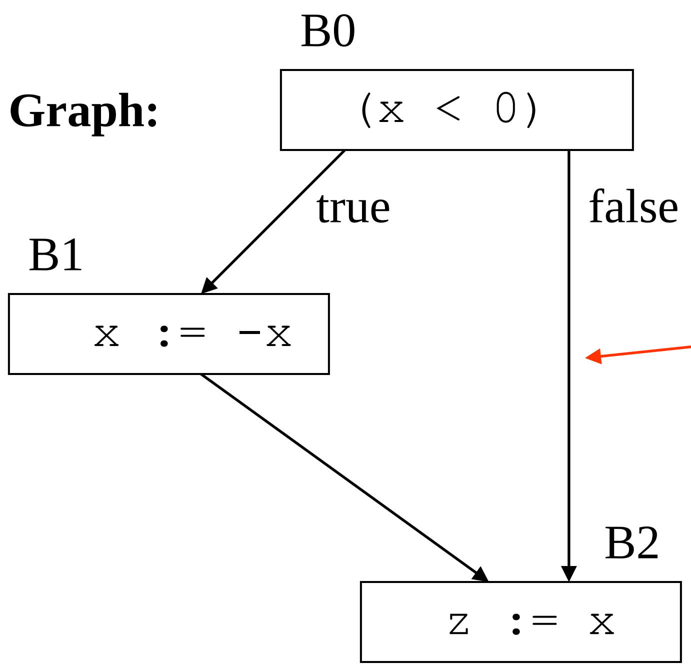
Test set {x=–1, x=2}gives both statement and branch coverage

Statement vs. Branch Coverage

```
assignAbsolute(int x)
{
  if (x < 0)
    x := -x;
  z := x;
}
```

Consider this program segment, the test set **T = {x = –1}** will give statement coverage, however not branch coverage

Control Flow Graph:



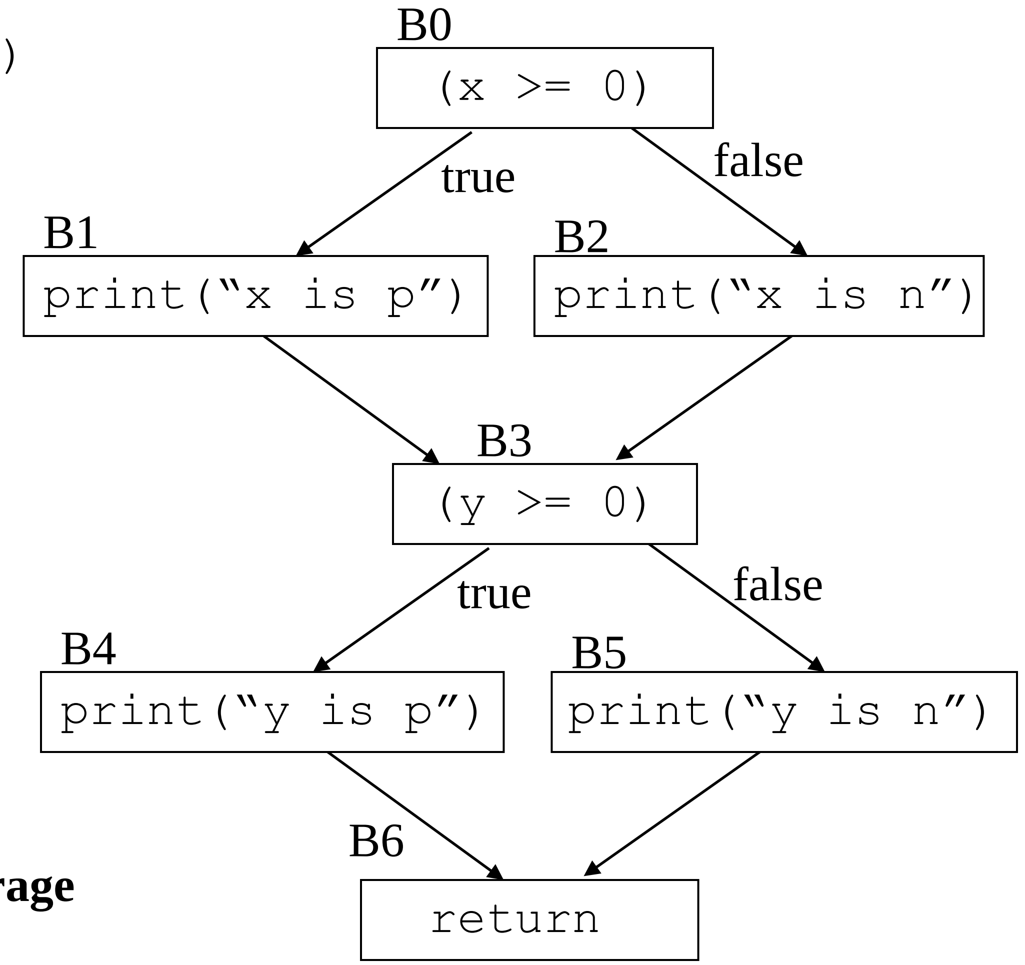
Test set **{x = –1}** does not execute this edge, hence, it does not give branch coverage

Path Coverage

- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, all paths leading from the initial to the final node of P's control flow graph are traversed

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```



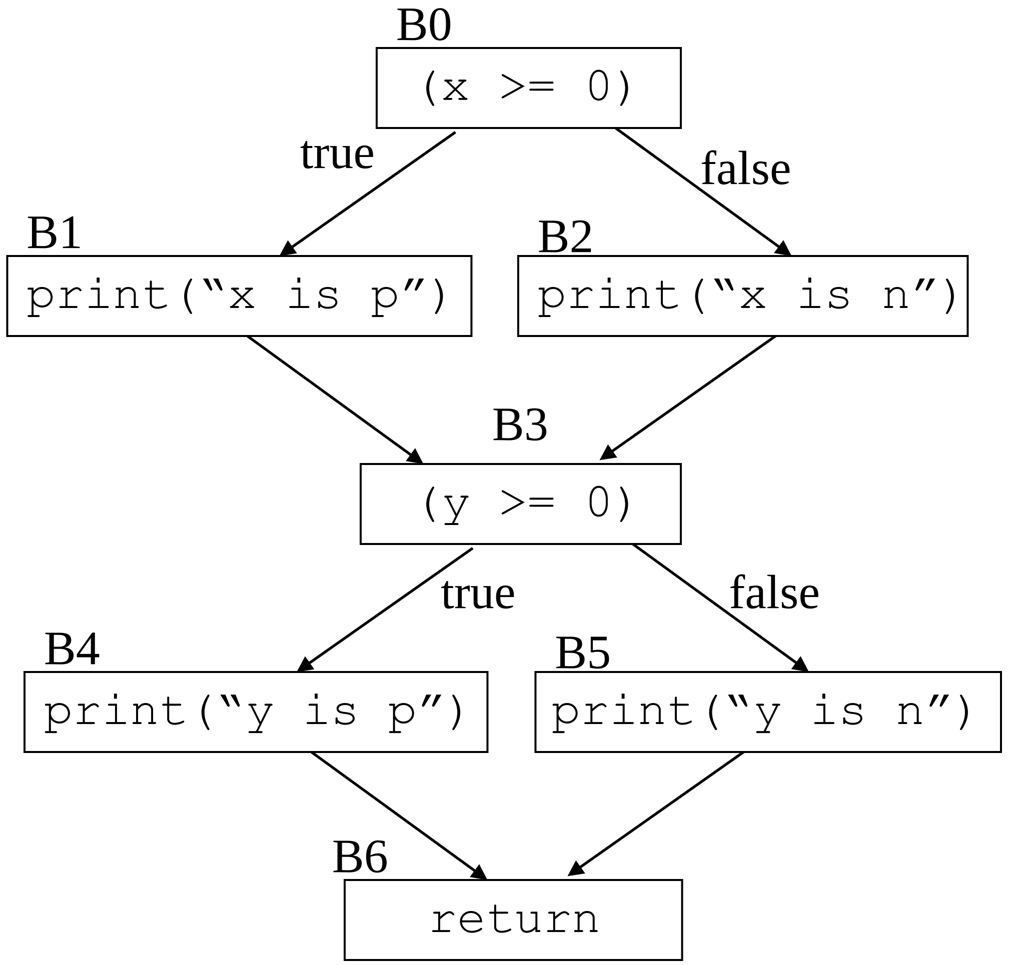
Test set:
T_2 = {(x = 12,y = – 5), (x = – 1,y = 35)}
gives both branch and statement coverage but it does not give path coverage

Set of all execution paths: {(B0,B1,B3,B4,B6), (B0,B1,B3,B5,B6), (B0,B2,B3,B4,B6), (B0,B2,B3,B5,B6)}
Test set T_2 executes only paths: (B0,B1,B3,B5,B6) and (B0,B2,B3,B4,B6)

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print(“x is positive”);
  else
    print(“x is negative”);
  if (y >= 0)
    print(“y is positive”);
  else
     print(“y is negative”);
}
```

Test set:
T_1 = {(x=12,y=5), (x=–1,y=35), (x=115,y=–13),(x=–91,y=–2)}
gives both branch, statement and path coverage



Cyclomatic Complexity

- Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it.
- It is a software metric used to indicate the complexity of a program.
- It is computed using the Control Flow Graph of the program.
- The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if second command might immediately follow the first command.
- ***M = E – N + 2***
- *where,*
  E = the number of edges in the control flow graph
  N = the number of nodes in the control flow graph

Cyclomatic Complexity

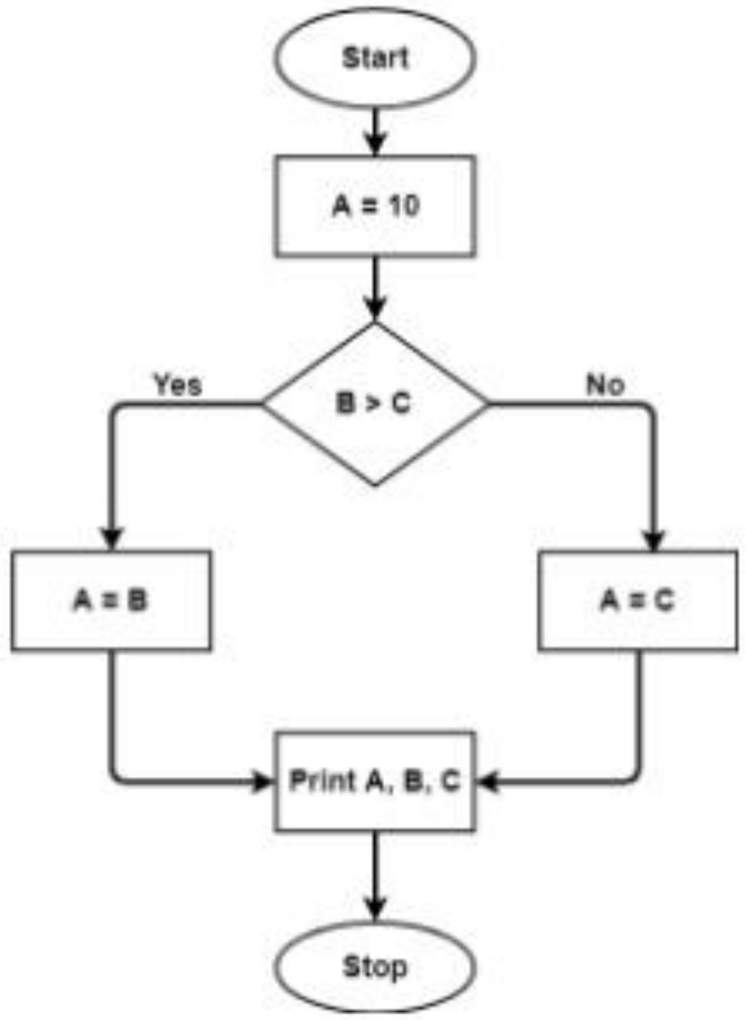
- Steps that should be followed in calculating cyclomatic complexity and test cases design are:

- Construction of graph with nodes and edges from code.
- Identification of independent paths.
- Cyclomatic Complexity Calculation
- Design of Test Cases

Cyclomatic Complexity

```
A = 10
  IF B > C THEN
    A = B
  ELSE
    A = C
  ENDIF
Print A
Print B
Print C
```

Cyclomatic Complexity



Cyclomatic Complexity

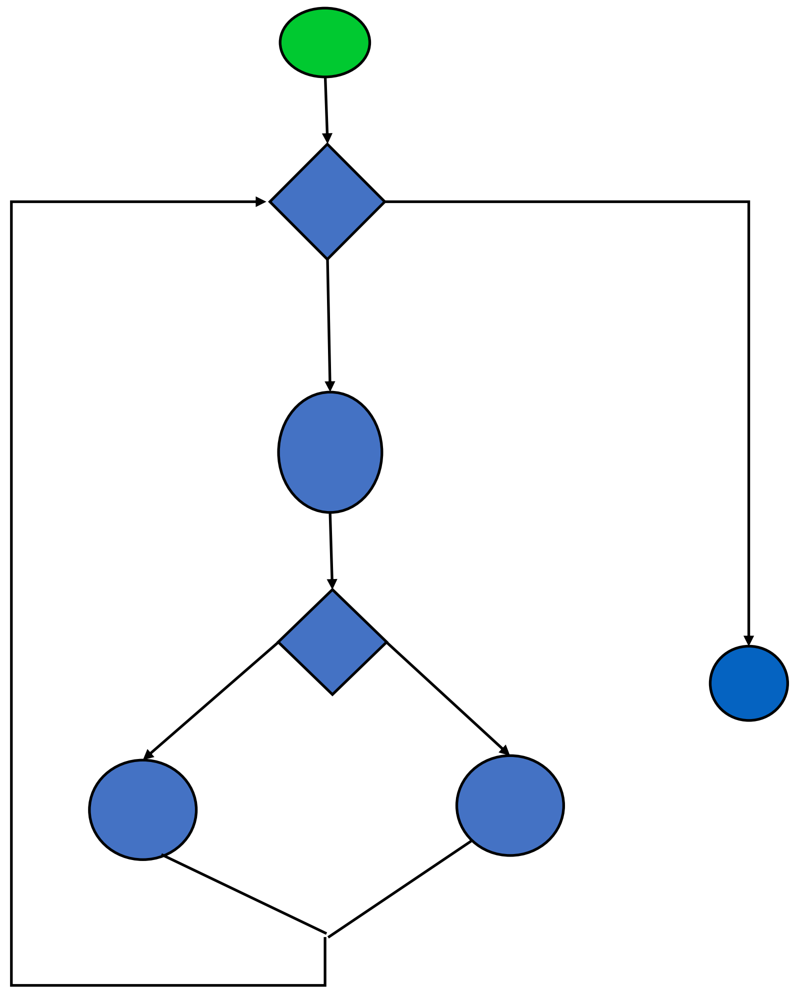
- The cyclomatic complexity calculated for above code will be from control flow graph. The graph shows seven shapes(nodes), seven lines(edges), hence cyclomatic complexity is 7-7+2 = 2.

Cyclomatic Complexity

```
public void howComplex() {
        int i=20;

        while (i<10) {
            System.out.printf("i is %d", i);
            if (i%2 == 0) {
                System.out.println("even");
            } else {
                System.out.println("odd");
            }
        }
    }
```

Cyclomatic Complexity

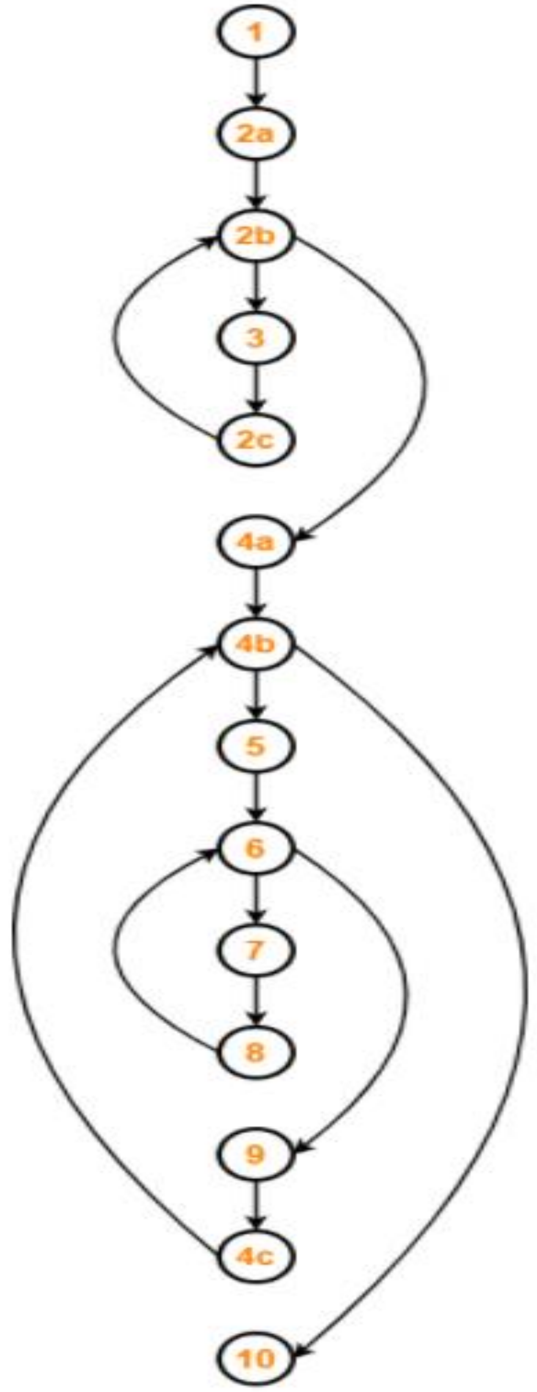


$V=E-N+2$

$V=8-7+2=3$

```
{ int i, j, k;
for (i=0 ; i<=N ; i++)
p[i] = 1;
for (i=2 ; i<=N ; i++)
{
k = p[i]; j=1;
while (a[p[j-1]] > a[k] {
p[j] = p[j-1];
j--;
}
p[j]=k;
}
```

```
{ int i, j, k;
for (i=0 ; i<=N ; i++)
p[i] = 1;
for (i=2 ; i<=N ; i++)
{
k = p[i]; j=1;
while (a[p[j-1]] > a[k] {
p[j] = p[j-1];
j--;
}
p[j]=k;
}
```



Static testing

- Static testing is the form of software testing where you do not execute the code being examined.
- This technique could be called non-execution technique.
- It is primarily syntax checking of the code or manually reviewing the code, requirements documents, design documents etc. to find errors.

Static testing (Contd.)

- From the black box testing point of view, static testing involves reviewing requirements and specifications.
- This is done with an eye toward completeness or appropriateness for the task at hand.
- This is the verification portion of Verification and Validation.
- The fundamental objective of static testing technique is to improve the quality of the software products by finding errors in early stages of software development life cycle.

Static testing (Contd.)

- Following are the main **Static Testing techniques** used:
  - **Formal Technical Reviews**
  - **Walkthrough**
  - **Code Inspection**

Formal Technical Reviews

- Formal Technical Reviews are conducted by software engineers.
- The primary objective is to find errors during the process so that they do not become defects after release of software as they uncover errors in function, logic design, or implementation.
- The idea is to have early discovery of errors so they do not propagate to the next step in the process.
- They also ensure that the software has been represented according to predefined standards and it is developed in a uniform manner.
- They make projects more manageable and help groom new resources as well as provide backup and continuity.

Formal Technical Reviews (Contd.)

- FTRs are usually conducted in a meeting that is successful only if it is properly planned, controlled.
- The producer informs the Project Manager that the Work Product is ready and the review is needed.
- The review meeting consists of 3-5 people and advanced preparation is required.
- It is important that this preparation should not require more than 2 hours of work per person.
- It should focus on specific (and small) part of the overall software.
- For example, instead of the entire design, walkthroughs are conducted for each component, or small group of components. By narrowing focus, FTR has a high probability of uncovering errors.

Formal Technical Reviews (Contd.)

- It is important to remember that the focus is on a work product (WP) for which the producer of the WP asks the project leader for review.
- Project leader informs the review leader.
- The review leader evaluates the WP for readiness and if satisfied generates copies of review material and distributes to reviewers for advanced preparation.

Formal Technical Reviews Review Meetings (Contd.)

- Review meeting (RM) is attended by the review leader, all reviewers, and the producer.
- One of the reviewer takes the roles of recorder.
- Producer walks through the product, explaining the material while other reviewers raise issues based upon their advanced preparation.
- When valid problems or errors are recorded, the recorder notes each one of them.
- At the end of the RM, all attendees of the meeting must decide whether to: Accept the product without further modification

Formal Technical Reviews Review Meetings (Contd.)

- Reject the product due to severe errors – Major errors identified – Must review again after fixing
- Accept the product provisionally – Minor errors to be fixed – No further review

Formal Technical Reviews

Review Reporting and Record keeping(Contd.)

- During the FTR the recorder notes all the issues.
- They are summarized at the end and a review issue list is prepared.
- A summary report is produced that includes:
  - What is reviewed
  - Who reviewed it
  - What were the findings and conclusions
- It then becomes part of project historical record.

Formal Technical Reviews (Contd.)

The review issue list

- It is sometimes very useful to have a proper review issue list. It has two objectives.
  - Identify problem areas within the WP Action item checklist
- It is important to establish a follow-up procedure to ensure that items on the issue list have been properly addressed.

Review Guidelines

- It is essential to note that an uncontrolled review can be worse than no review.
- The basis principle is that the review should focus on the product and not the producer so that it does not become personal.
- Errors should be pointed out gently and the tone should be loose and constructive.

Walkthrough

- A walkthrough is a term describing the consideration of a process at an abstract level.
- The term is often employed in the software industry (see software walkthrough) to describe the process of inspecting algorithms and source code by following paths through the algorithms or code as determined by input conditions and choices made along the way.
- The purpose of such code walkthroughs is generally to provide assurance of the fitness for purpose of the algorithm or code; and occasionally to assess the competence or output of an individual or team.

Guidelines of Walkthrough

- The team performing code walk through should not be either too big or too small.
- Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code Inspection

- The aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
- During code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs.
- For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter Errors.

Code Inspection (Contd.)

- It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure.
- In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.
- Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed.
- Such a list of commonly committed errors can be used during code inspection to look out for possible

Code Inspection (Contd.)

- Following is a list of some classical programming errors which can be checked during code inspection:
    - Use of un-initialized variables.
    - Jumps into loops.
    - Non-terminating loops.
    - Incompatible assignments.
    - Array indices out of bounds.
    - Improper storage allocation and de-allocation.
    - Mismatches between actual and formal parameter in procedure calls.
    - Use of incorrect logical operators or incorrect precedence among operators.
    - Improper modification of loop variables.
    - Comparison of equally of floating point variables, etc.

Coding

- The goal of coding phase is to translate the design of the system into code in a given programming language, which can be executed by a computer and that performs the computation specified by the design.
- A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.
- A coding standard gives a uniform appearance to the codes written by different engineers.
- It enhances code understanding.
- It encourages good programming practices.

Importance of Good Programming Style

To simplify software maintenance

To avoid problems

To simplify the testing process.

To achieve readability.

To improve modifiability

To improve transportability

To make a robust product

Coding standards and guidelines

- **Rules for limiting the use of global: These rules list what types of data can be** declared global and what cannot.
- **Contents of the headers preceding codes for different modules:** The following are some standard header data:
  - Name of the module.
  - Date on which the module was created.
  - Author’s name.
  - Modification history.
  - Synopsis of the module.
  - Different functions supported, along with their input/output parameters.
  - Global variables accessed/modified by the module.

Coding standards and guidelines

- **Naming conventions for global variables, local variables, and constant identifiers: A possible naming convention can be that global variable names** always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
- **Error return conventions and exception handling mechanisms: The way** error conditions are reported by different functions in a program are handled should be standard within an organization.

Coding standards and guidelines: Recommended by SW Development Organization

- **Do not use a coding style that is too clever or too difficult to understand**
- **Do not use an identifier for multiple purposes**
- **The code should be well-documented**
- **The length of any function should not exceed 10 source lines**
- **Do not use goto statements**
- **Do not do hard coding**