SQA : UNIT 2

NEP 24-25

UNIT 2 : Syllabus

- **2.1 Unit Testing**: Black Box Testing
- 2.1.1 Boundary Value Analysis and Testing: Normal Boundary Value Testing, Special Value Testing, Examples, Random Testing, Guidelines for Boundary Value Testing, Non-functional Boundaries,
- Functional Boundaries
- 2.1.2 Equivalence Class Testing: Equivalence Classes, Traditional Equivalence Class Testing, Avoiding Equivalence Partitioning Errors, Composing Test Cases with Equivalence Partitioning, Equivalence Partitioning Exercise, Examples of Equivalence Partitioning and Boundary Values, Edge Testing, Guidelines and Observations.
- 2.1.3 Decision Table–Based Testing: Decision Tables, Decision Table
- Techniques, Cause-and-Effect Graphing, Guidelines and Observations,

- **2.2 Path Testing**: White Box Testing, Program Graphs, DD-Paths,

Test Coverage Metrics, Basis Path, Testing, Guidelines and Observations, Data Flow Testing: Define/Use Testing, Slice-Based Testing, Program Slicing Tools.

- **2.3 Software Verification and Validation**: Introduction, Verification, Verification Workbench, Methods of Verification, Entities involved in verification, Reviews in testing lifecycle, Coverage in Verification, Concerns of Verification, Validation, Validation Workbench, Levels of Validation, Coverage in Validation, Acceptance Testing, Management of Verification and Validation,
- **2.4 V-test Model**: Introduction, V-model for software, testing during
- Proposal stage, Testing during requirement stage, Testing during test
- planning phase, Testing during design phase, Testing during coding,
- VV Model, Critical Roles and Responsibilities.

Introduction

- Black Box Testing: Use to perform ‘correctness tests’ and more
- Some of these white box tests, such as maintainability tests and calculation correctness testing.
- Bottom line: BB testing canNOT substitute for WB testing.

- Points to consider :
  - Equivalence class testing and number of test cases required
  - Performance methodology for other classes of BB tests
  - Advantages and disadvantages of BB testing.

Black Box Testing

- Testing software against a specification of its external behavior without knowledge of internal implementation details
  - Can be applied to software “units” (e.g., classes) or to entire programs
  - External behavior is defined in API docs, Functional specs, Requirements specs, etc.

- Because black box testing purposely disregards the program's control structure, attention is focused primarily on the information domain (i.e., data that goes in, data that comes out)

- The Goal: Derive sets of input conditions (test cases) that fully exercise the external functionality

2.1.1 Boundary Value Testing

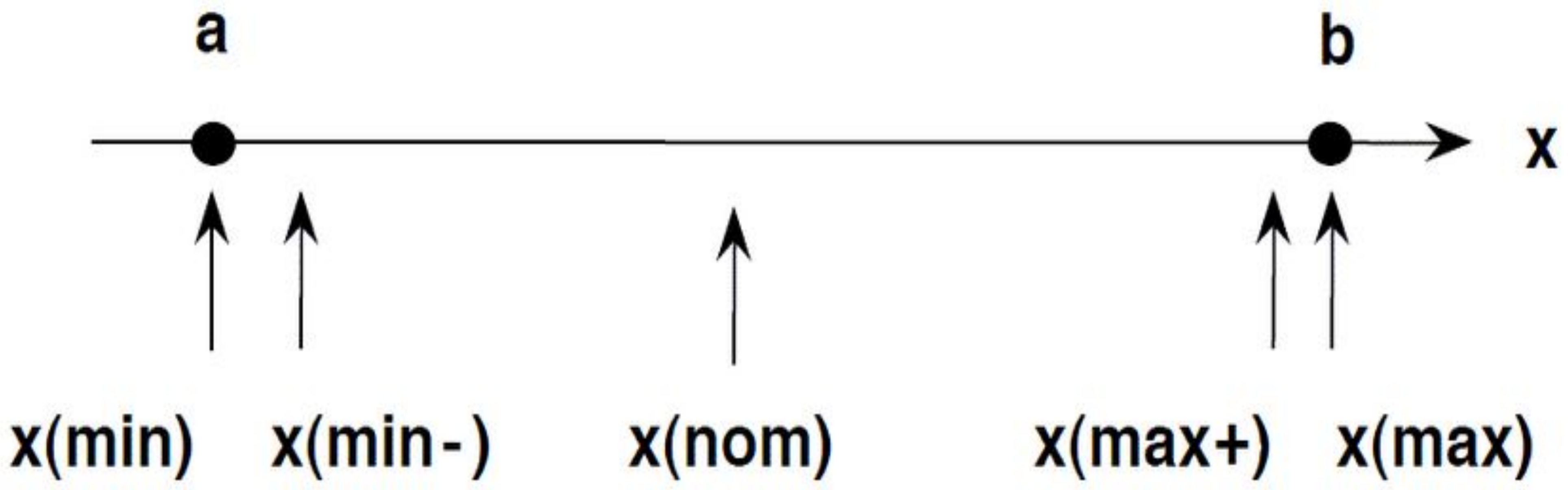
- In ST the Boundary Value Testing is a black box test design technique based on test cases.
- Input domain testing is also known as “BVT” is the best-known best-specification testing technique.
- Normal BVT is concerned only with valid value of the input variables.
- Robust boundary value testing considers invalid and valid variable values.

Types of Boundary Value Testing

- Normal Boundary Value Testing
- Robust Boundary Value Testing
- Worst-case Boundary Value Testing
- Robust Worst-case Boundary Value Testing

What is Boundary Testing?

- Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.
- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called “boundary testing”.
- The basic idea in normal boundary value testing is to select input variable values at their:
- Minimum
- Just above the minimum
- A nominal value
- Just below the maximum
- Maximum


a
b
x
x(min)
x(min-)
x(nom)
x(max+)
x(max)

Advantages of Boundary Value Testing

- *The BVA technique of testing is quite easy to use and remember because of the uniformity of identified tests and the automated nature of this technique.*
- *One can easily control the expenses made on the testing by controlling the number of identified test cases. This can be done with respect to the demand of the software that needs to be tested.*
- *BVA is the best approach in cases where the functionality of a software is based on numerous variables representing physical quantities.*
- *The technique is best at revealing any potential UI or user input troubles in the software.*
- *The procedure and guidelines are crystal clear and easy when it comes to determining the test cases through BVA.*
- *The test cases generated through BVA are very small.*
-

Disadvantages of Boundary Value Testing

- *This technique sometimes fails to test all the potential input values. And so, the results are unsure.*
- *The dependencies with BVA are not tested between two inputs.*
- *This technique doesn’t fit well when it comes to Boolean Variables.*
- *It only works well with independent variables that depict quantity.*

Normal Boundary Value Testing

- Normal Boundary value analysis is a black-box testing technique, closely associated with equivalence class partitioning.
- In this technique, we analyze the behavior of the application with test data residing at the boundary values of the equivalence classes.
- The basic idea in boundary value testing is to select input variable values at their:
- Minimum
- Just above the minimum
- A nominal value
- Just below the maximum
- Maximum
-

Advantages of boundary value analysis

- It is easier and faster to find defects as the density of defects at boundaries is more.
- The overall test execution time reduces as the number of test data greatly reduces.

Disadvantages of boundary value analysis

- The success of the testing using boundary value analysis depends on the equivalence classes identified, which further depends on the expertise of the tester and his knowldege of application. Hence, incorrect identification of equivalence classes leads to incorrect boundary vlaue testing.
- For application with open boundaries or application not having one dimensional boundaries are not suitable for boundary value analysis. In those cases, other black-box techniques like "Domain Analysis" are used.

Robust Boundary Value Testing

- Robust BVT technique like BVT technique , but this technique is extend negative case in boundary and still focus on Mind, Mid and Max.
- In this technique s/w is tested by giving invalid input or data.
- Robustness testing is usually done to test exception handling.
- In this techniques we make combinations in such a way that some of the invalid values are also tested as input.

Worst-case Boundary Value Testing

- Boundary Value analysis uses the critical fault assumption and therefore only tests for a single variable at a time assuming its extreme values.
- By disregarding this assumption we are able to test the outcome if more than one variable were to assume its extreme value.
- In an electronic circuit this is called Worst Case Analysis.
- In Worst-Case testing we use this idea to create test cases.

Robust Worst-case Boundary Value Testing

- If the function under test were to be of the greatest importance we could use a method named Robust Worst-Case testing which as the name suggests draws it attributes from Robust and Worst-Case testing.
- for increase more and more chance to find defect, the testing add +1 in max and -1 in min.
- Focus on worst-case robust boundary of the input space to identify test cases.
- Obviously this results in the largest set of test results we have seen so far and requires the most effort to produce.

Special Value Testing

- Special Value is defined and applied form of Functional Testing, which is a type of testing that verifies whether each function of the software application operates in conformance with the required specification.
- Special value testing is probably the most extensively practiced form of functional testing which is most intuitive and least uniform.
- This technique is performed by experienced professionals who are experts in this field and have profound knowledge about the test and data required for it.
- They continuously participate and apply tremendous efforts to deliver appropriate test results to suit the client’s requested demand.

Why Special Value Testing

- The testing executed by Special Value Testing technique is based on past experiences, which validates that no bugs or defects are left undetected.
- The testers are extremely knowledgeable about the industry and use this information while performing Special Value Testing.
- Another benefits of opting Special Value Testing technique is that it is ad-hoc in nature
- There are no guidelines used by the testers other that their “Best engineering judgment”.
- The most important aspect of this testing is that, it has had some very valuable inputs and success in finding bugs and errors while testing a software.

Random Testing

- **Random testing** is a black-box software testing technique where programs are tested by generating random, independent inputs.
- Results of the output are compared against software specifications to verify that the test output is pass or fail.
- In case of absence of specifications the exceptions of the language are used which means if an exception arises during test execution then it means there is a fault in the program, it is also used as way to avoid biased testing.
- andom testing is performed where the defects are NOT identified in regular intervals.
- Random input is used to test the system's reliability and performance.
- Saves time and effort than actual test efforts.
- Other Testing methods Cannot be used to.

Monkey Testing

- Monkey testing is a software testing technique in which the testing is performed on the system under test randomly.
- In Monkey Testing the tester (sometimes developer too) is considered as the 'Monkey'
- If a monkey uses a computer he will randomly perform any task on the system out of his understanding
- Just like the tester will apply random test cases on the system under test to find bugs/errors without predefining any test case
- In some cases, Monkey Testing is dedicated to unit testing
  This testing is so random that the tester may not be able to reproduce the error/defect.
- The scenario may NOT be definable and may NOT be the correct business case.
- Monkey Testing needs testers with very good domain and technical expertise.

The advantages of Monkey testing:

- As the scenarios that are tested are adhoc, system might be under stress so that we can also check for the server responses.
- This testing is adopted to complete the testing, in particular if there is a resource/time crunch.

The Disadvantages of Monkey testing

- No bug can be reproduced: As tester performs tests randomly with random data reproducing any bug or error may not be possible.
- Less Accuracy: Tester cannot define exact test scenario and even cannot guarantee the accuracy of test cases
- Requires very good technical expertise: It is not worth always to compromise with accuracy, so to make test cases more accurate testers must have good technical knowledge of the domain
- Fewer bugs and time consuming: This testing can go longer as there is no predefined tests and can find less number of bugs which may cause loopholes in the system

Guidelines for Boundary Value Testing,

- With the exception of special value testing, the test methods based on the boundary values of a program are the most rudimentary.
- Issues in producing satisfactory test cases using boundary value testing:
- Truly independent variables versus not independent variables
- Normal versus robust values
- Single fault versus multiple fault assumption
- Boundary value analysis can also be applied to the output range of a program (i.e. error messages), and internal variables (i.e. loop control variables, indices, and pointers).

2.1.2 Equivalence Class Testing

- Equivalence Classes, Traditional Equivalence Class Testing, Improved Equivalence Class Testing,
- Avoiding Equivalence Partitioning Errors, Composing Test Cases with Equivalence Partitioning, Equivalence Partitioning Exercise, Examples of Equivalence Partitioning and Boundary Values , Edge Testing, Guidelines and Observations.

Equivalence Class Testing

- Equivalence Partitioning also called as equivalence class partitioning.
- It is abbreviated as ECP.
- This technique used by the team of testers for grouping and partitioning of the test input data, which is then used for the purpose of testing the software product into a number of different classes.
- It is a software testing technique that divides the input test data of the application under test into each partition at least once of equivalent data from which test cases can be derived.
- An advantage of this approach is it reduces the time required for performing testing of a software due to less number of test cases.

Advantages Equivalence Class Testing

- It helps reduce the number of test cases, without compromising the test coverage.
- Reduces the overall test execution time as it minimizes the set of test data
- It can be applied to all levels of testing, such as unit testing, integration testing, system testing etc.
- Enables the testers to focus on smaller data sets, which increases the probability to uncovering more defects in the software product.
- It is used in cses where performing exhaustive testing is difficult but at the same time maintaining good coverage is required.

Disadvantages Equivalence Testing

- It does not consider the conditions for boundary value.
- The identification of equivalence classes relies heavily on the expertise of testers.
- Testers might assume that the o/p for all i/p data set are correct, which can become a great hurdle in testing.

Types of Equivalence Class Testing

- Weak normal equivalence class testing: Uses one valid input variable from every single equivalence class for a test case & follows the single fault assumption
- Strong normal equivalence class testing: Uses the Cartesian product of the equivalence classes of each valid input variables to obtain the test case & follows the multiple fault assumption
- Weak robust equivalence class testing: Defines equivalence class in terms of the class of valid inputs & class of invalid inputs for test case.
- Strong robust equivalence class testing: Uses every single element of Cartesian product of all the equivalence classes to acquire test cases.

Strong Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- Test cases from Cartesian Product of valid values.
- Detects faults due to interactions with valid values of any number of variables.
- OK for regression testing, better for progression testing.

Weak Robust Equivalence Class Testing

- Identify equivalence classes of valid and invalid values.
- Test cases have all valid values except one invalid value.
- Detects faults due to calculations with valid values of a single variable.
- Detects faults due to invalid values of a single variable.
- OK for regression testing

Weak Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- Test cases have all valid values.
- Detects faults due to calculations with valid values of a single variable.
- OK for regression testing.
- Need an expanded set of valid classes

Strong Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- Test cases from Cartesian Product of valid values.
- Detects faults due to interactions with valid values of any number of variables.
- OK for regression testing, better for progression testing.

Improved Equivalence Class Testing

- There are two main properties that underpin the methods used in functional testing.
- These two properties lead to two different types of equivalence class testing, weak and strong.
- However if one decide to test for invalid i/p or o/p as well as valid i/p or o/p we can produce another two different types of equivalence class testing, normal and robust.
- Robust equivalence class testing takes into consideration the testing of invalid values, whereas normal does not.

Equivalence class partitioning (EC)

A black box method aimed at increasing the efficiency of testing and, at the same time, improving coverage of potential error conditions.

Equivalence Partitioning

- **Equivalence Partitioning** or Equivalence Class Partitioning is type of black box testing technique which can be applied to all levels of software testing like unit, integration, system, etc.
- In this technique, input data units are divided into equivalent partitions that can be used to derive test cases which reduces time required for testing because of small number of test cases.
- It divides the input data of software into different equivalence data classes.
- You can apply this technique, where there is a range in the input field.

Equivalence class partitioning (EC)

- **A**n **equivalence class** (EC) is a set of input variable values that produce the same output results or that are processed identically.
  - Again, consider a range of acceptable values: say, 1 – 10.
  - Tests with inputs 2 through 9 should produce the same result.
  - No need for more than one test with input value 2 to 9.
- **EC <u>boundaries</u>** are defined by a single numeric or alphabetic value, a group of numeric or alphabetic values, a range of values, and so on.
  - If range is 1-10, boundaries would be 1 and 10.

Equivalence class partitioning (EC)

- An EC with only valid states is defined as a "**valid EC**," whereas an EC that contains only invalid states is defined as the "**invalid EC**."
  - **By definition!**

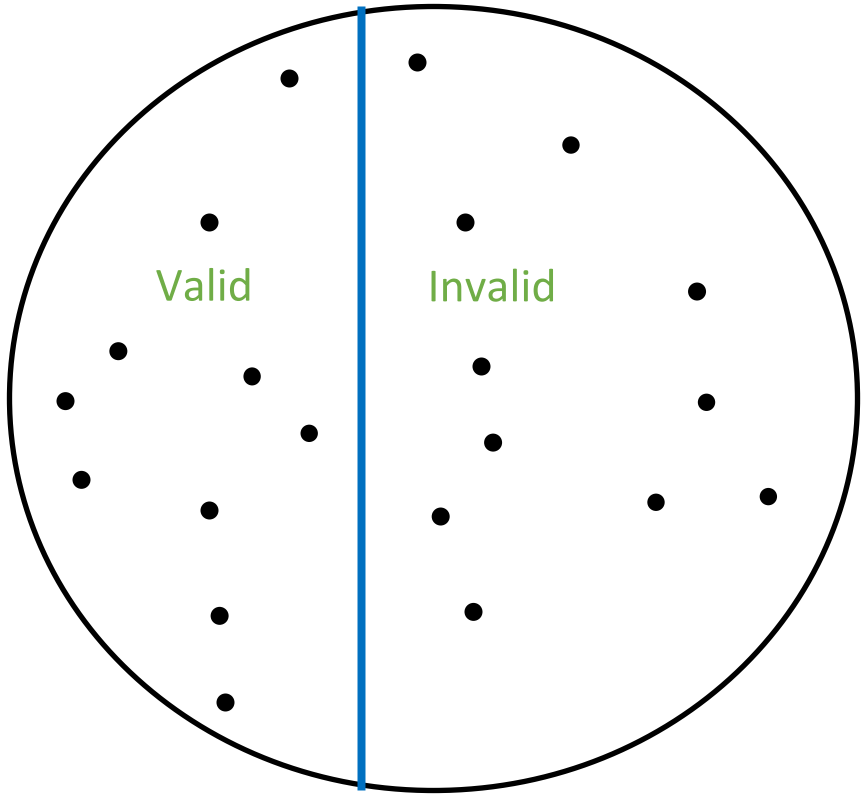
- In cases where a program's input is provided by several variables, valid and invalid ECs should be defined for each variable.

Equivalence Partitioning

- Partition the test cases into "equivalence classes"
- Each equivalence class contains a set of "equivalent" test cases
- Two test cases are considered to be equivalent if we expect the program to process them both in the same way (i.e., follow the same path through the code)
- If you expect the program to process two test cases in the same way, only test one of them, thus reducing the number of test cases you have to run

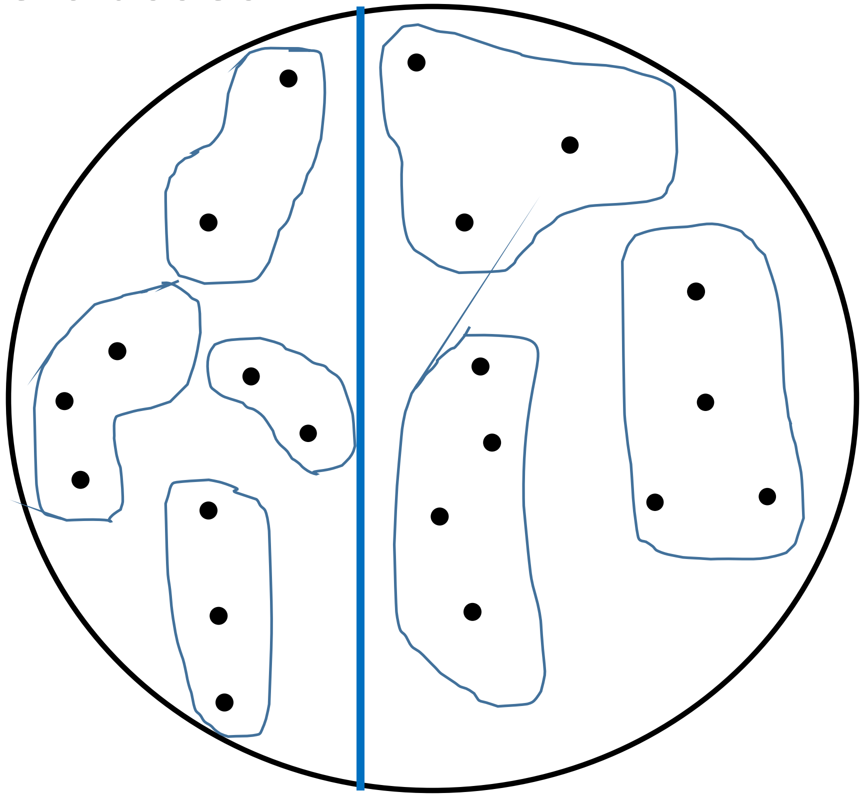
Equivalence Partitioning

- First-level partitioning: Valid vs. Invalid test cases



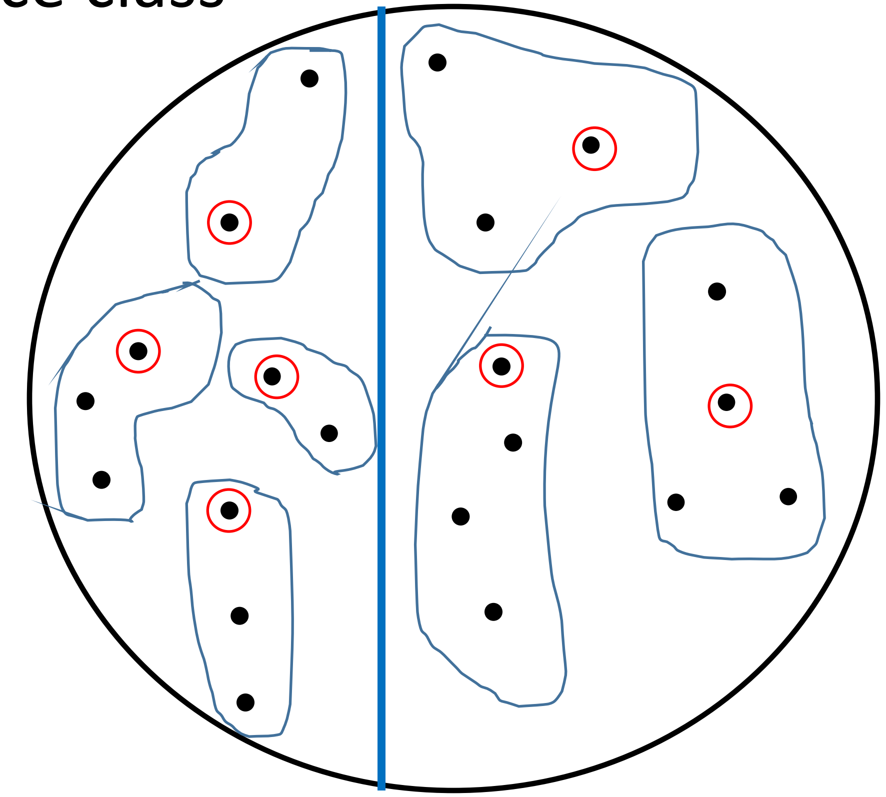
Equivalence Partitioning

- Partition valid and invalid test cases into equivalence classes



Equivalence Partitioning

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



Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

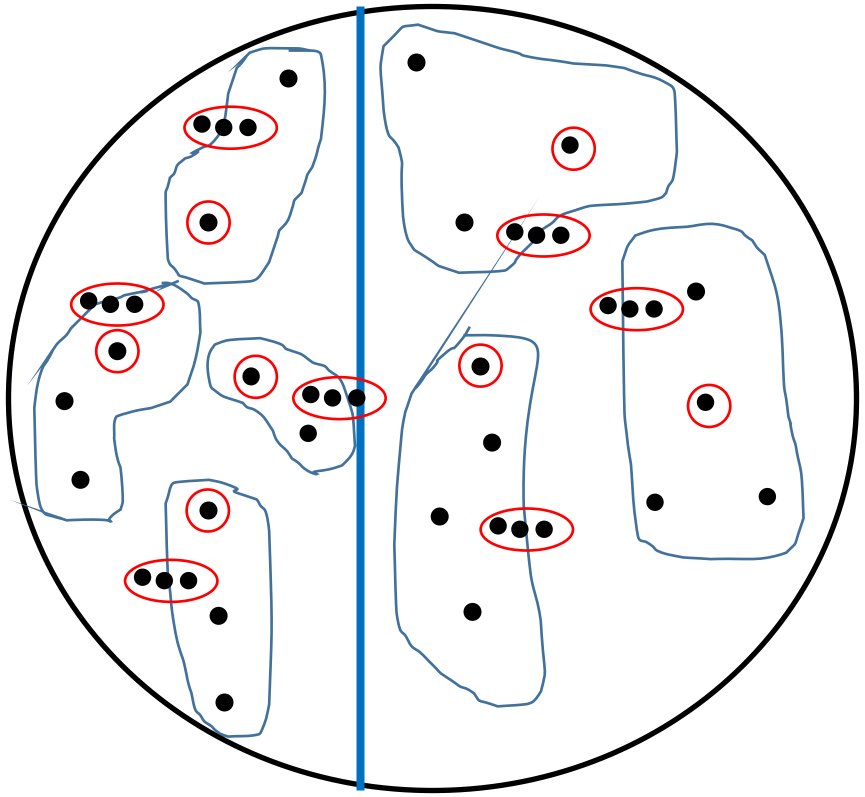
Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, …} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	Invalid format 5555555, (555)(555)5555, etc. Area code < 200 or > 999 Area code with non-numeric characters *Similar for Prefix and Suffix*

Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: -99 <= N <= 99	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: -99 <= N <= 99	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

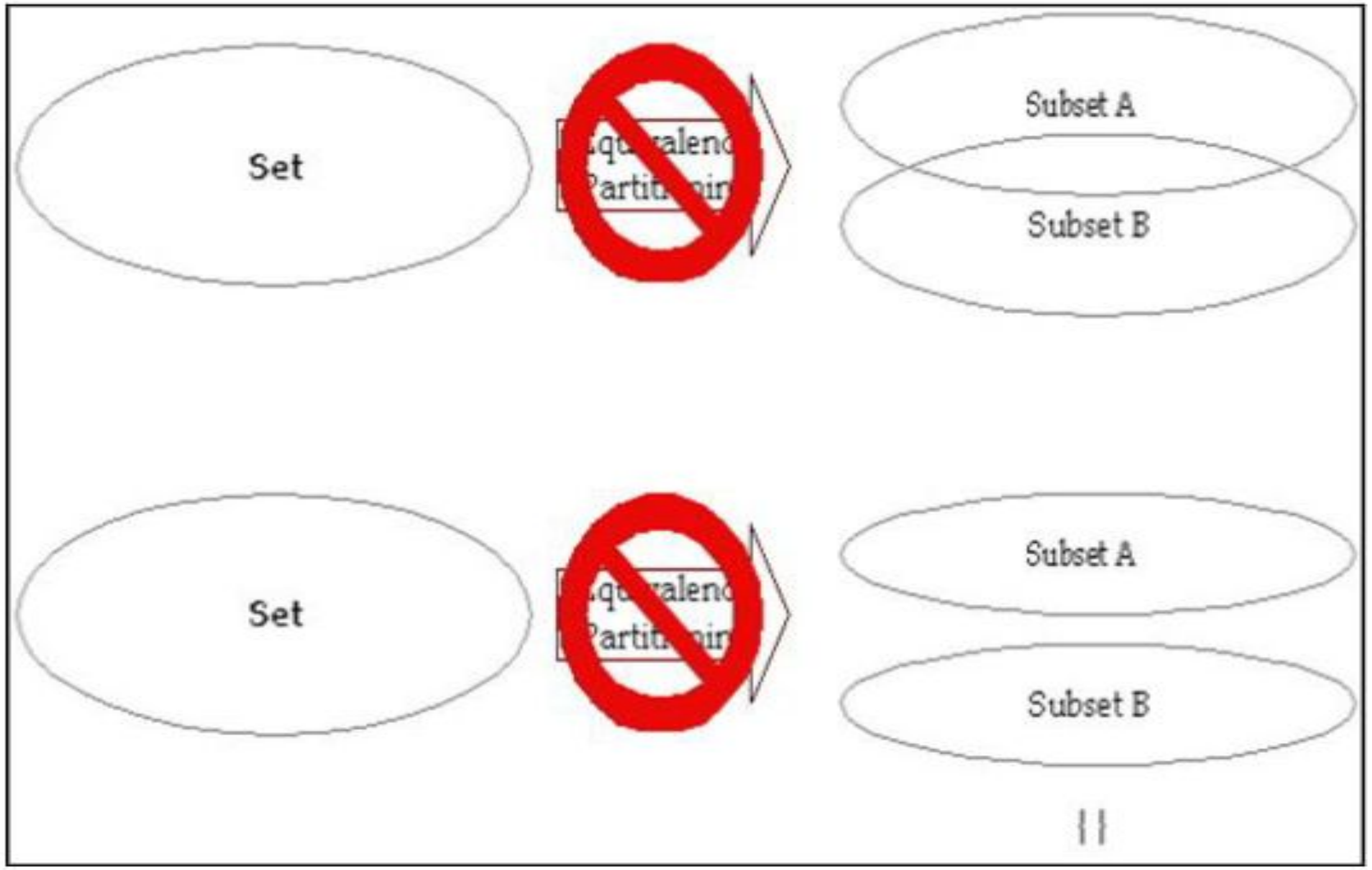
Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that:	-100, -99, -98
-99 <= N <= 99	-10, -9
	-1, 0, 1
	9, 10
	98, 99, 100
Phone Number	Area code: 199, 200, 201
Area code: [200, 999]	Area code: 998, 999, 1000
Prefix: (200, 999]	Prefix: 200, 199, 198
Suffix: Any 4 digits	Prefix: 998, 999, 1000
	Suffix: 3 digits, 5 digits

Avoiding Equivalence Partitioning Errors

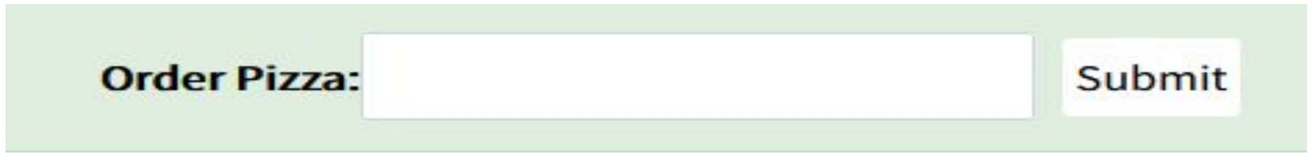
1. The subsets must be disjoint. That is, no two of the subsets can have one or more members in common. (The whole point of equivalence partitioning is to test whether a system handles different situations differently)
2. None of the subsets may be empty. (If the equivalence partitioning operation produces a subset with no members, that's hardly very useful for testing.) We can't select a member of that subset, because it has no members.
3. The equivalence partitioning process does not subtract, it divides.

Avoiding Equivalence Partitioning Errors



Example 1: Equivalence and Boundary Value

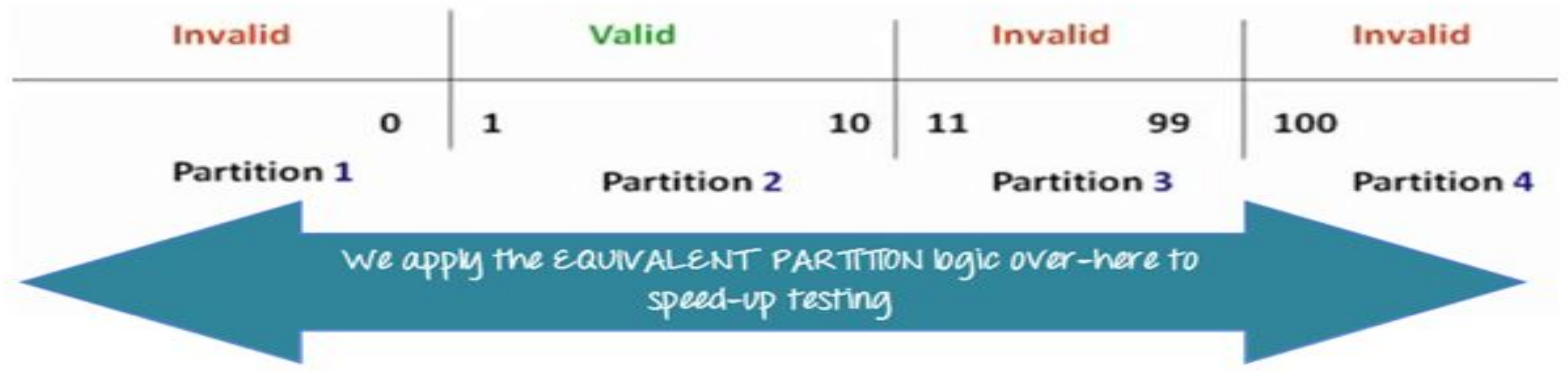
- Let's consider the behavior of Order Pizza Text Box Below
- Pizza values 1 to 10 is considered valid. A success message is shown.
- While value 11 to 99 are considered invalid for order and an error message will appear, **"Only 10 Pizza can be ordered"**



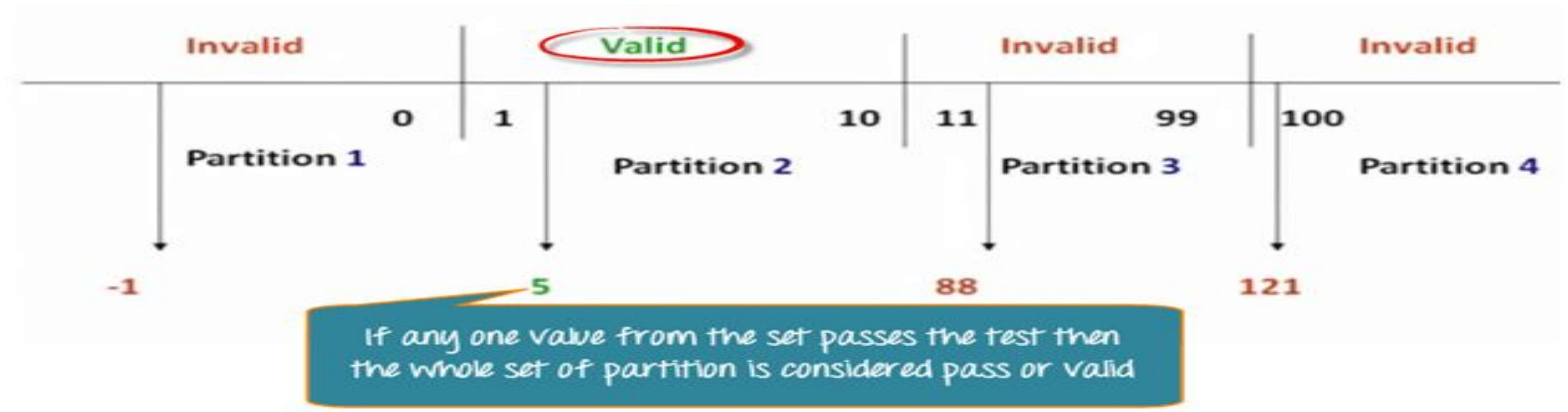
- **Here is the test condition**

1. Any Number greater than 10 entered in the Order Pizza field(let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.

- We cannot test all the possible values because if done, the number of test cases will be more than 100. To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same.

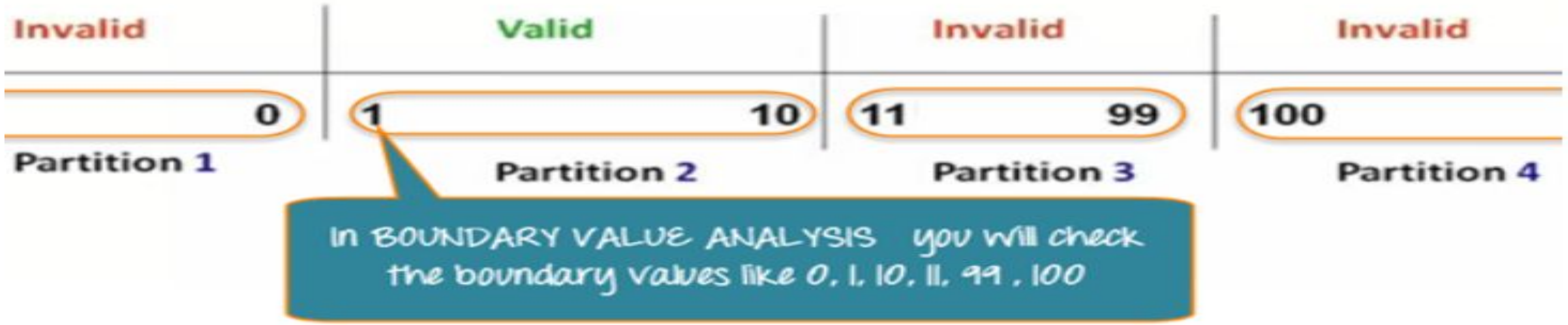


- The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is **that if one condition/value in a partition passes all others will also pass**. Likewise**, if one condition in a partition fails, all other conditions in that partition will fail**.


Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
-1
5
88
121
If any one value from the set passes the test then the whole set of partition is considered pass or valid

Boundary Value Analysis

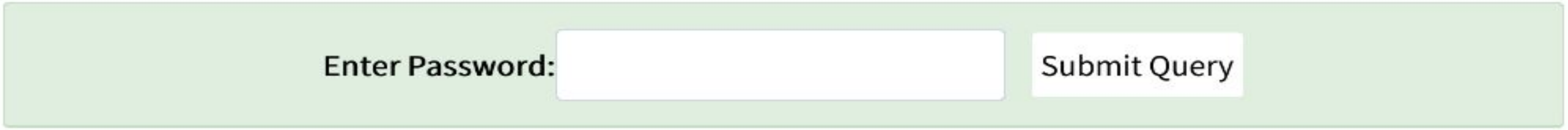
- In Boundary Value Analysis, you test boundaries between equivalence partitions

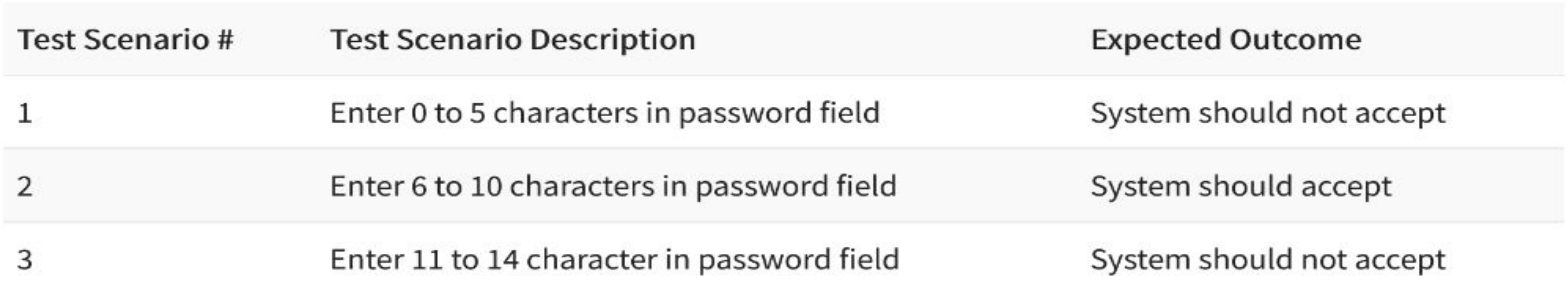

Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99, 100

- In our earlier equivalence partitioning example, instead of checking one value for each partition, you will check the values at the partitions like 0, 1, 10, 11 and so on. As you may observe, you test values at **both valid and invalid boundaries**. Boundary Value Analysis is also called **range checking**.
- Equivalence partitioning and boundary value analysis(BVA) are closely related and can be used together at all levels of testing.

Example 2: Equivalence and Boundary Value

- Following password field accepts minimum 6 characters and maximum 10 characters
- That means results for values in partitions 0-5, 6-10, 11-14 should be equivalent

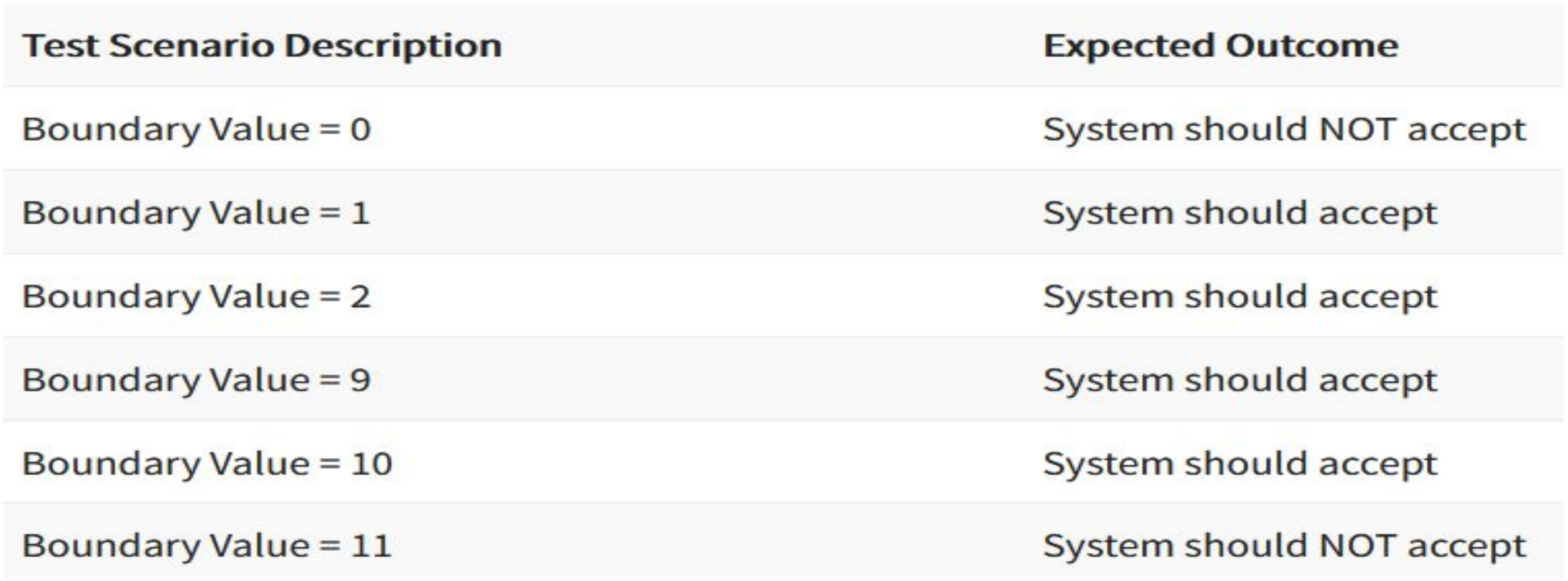




Test Scenario #	Test Scenario Description	Expected Outcome
1	Enter 0 to 5 characters in password field	System should not accept
2	Enter 6 to 10 characters in password field	System should accept
3	Enter 11 to 14 character in password field	System should not accept

Example 3: Input Box should accept the Number 1 to 10

- Here we will see the Boundary Value Test Cases



Test Scenario Description	Expected Outcome
Boundary Value = 0	System should NOT accept
Boundary Value = 1	System should accept
Boundary Value = 2	System should accept
Boundary Value = 9	System should accept
Boundary Value = 10	System should accept
Boundary Value = 11	System should NOT accept

Why Equivalence & Boundary Analysis Testing

- This testing is used to reduce a very large number of test cases to manageable chunks.
- Very clear guidelines on determining test cases without compromising on the effectiveness of testing.
- Appropriate for calculation-intensive applications with a large number of variables/inputs
- **Summary:**
- Boundary Analysis testing is used when practically it is impossible to test a large pool of test cases individually
- Two techniques - Boundary value analysis and equivalence partitioning testing techniques are used
- In Equivalence Partitioning, first, you divide a set of test condition into a partition that can be considered.
- In Boundary Value Analysis you then test boundaries between equivalence partitions
- Appropriate for calculation-intensive applications with variables that represent physical quantities

Edge Testing, Guidelines and Observations.

- Edge test case scenarios are those that are possible, but unknown or **accidental features** of the requirements.
- Boundary testing, in which testers validate between the extreme ends of a range of inputs, is a great way to find edge cases when testers are dealing with specific and calculated value fields.
- Edge case testing, or fringe testing, assesses how the application works when you go off script.
- Fringe testing can include testing out of sequence, twisting the expected workflow, or discovering functions the application performs that it was not designed for.
- When you test edge cases, you're using creativity to get the application to perform functions it's not intended to do, and figuring out the results.

2.1.3 Decision Table–Based Testing

- Decision Tables, Decision Table Techniques, Cause-and-Effect Graphing, Guidelines and Observations,

Decision Tables

- Decision tables are **a concise visual representation for specifying which actions to perform depending on given conditions**.
- The information expressed in decision tables could also be represented as decision trees or in a programming language as a series of if-then-else and switch-case statements.
- **Decision tables** are a precise yet compact way to model complicated logic.
- Decision tables, like if-then-else and switch-case statements, associate conditions with actions to perform.
- But, unlike the control structures found in traditional programming languages, decision tables can associate many independent conditions with several actions in an elegant way.

Decision Tables - Usage

- Decision tables make it easy to observe that all possible conditions are accounted for.

- Decision tables can be used for:
  - Specifying complex program logic
  - Generating test cases (Also known as *logic-based testing)*

- *Logic-based testing* is considered as:
  - <u>structural testing</u> when applied to structure (i.e. control flowgraph of an implementation).
  - <u>functional testing</u> when applied to a specification.

The general format of a decision table has four basic parts as shown below.

1. Action entry:

It indicates the actions to be taken.

2. Condition entry:

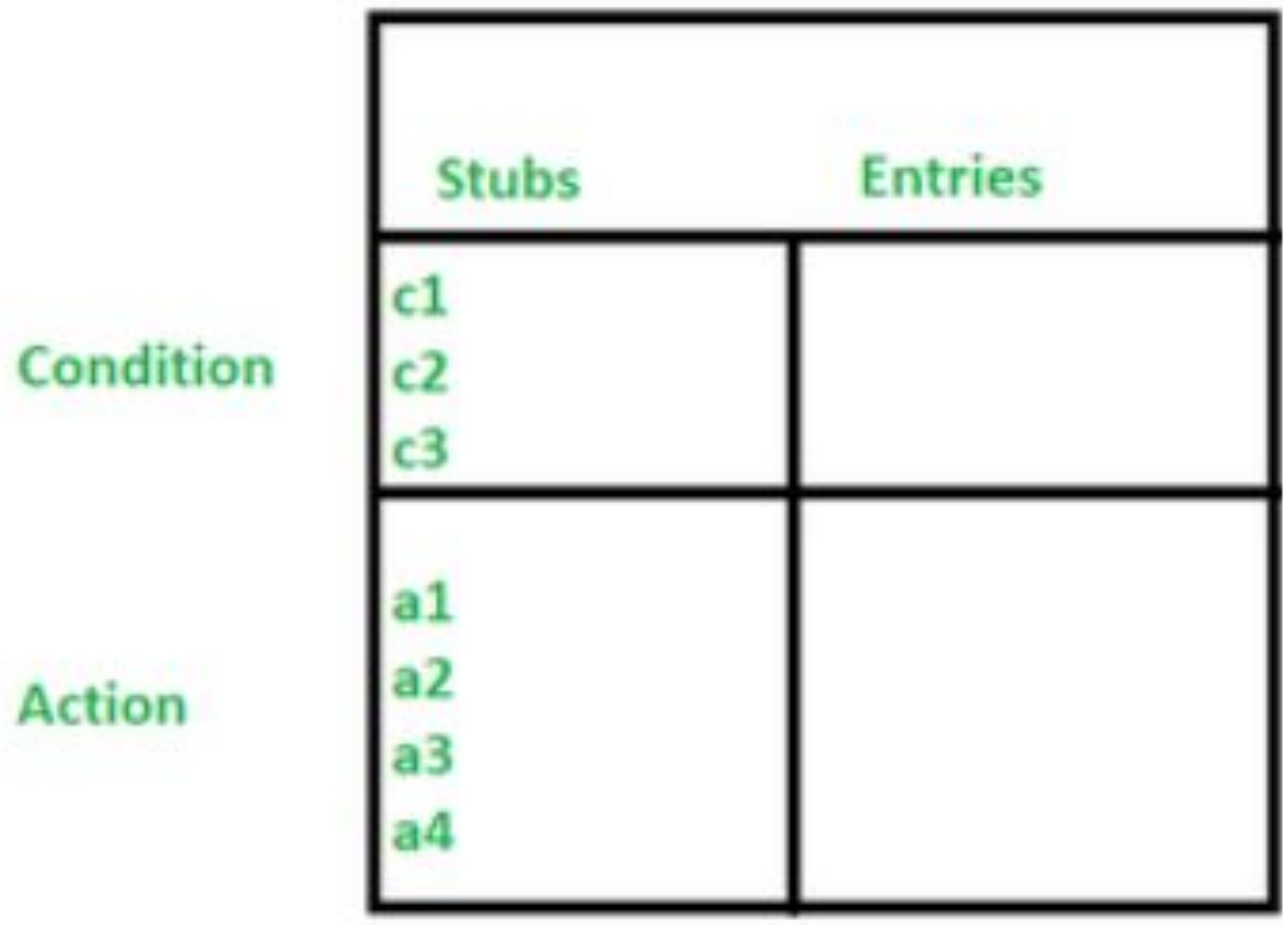
It indicates conditions which are being met or answers the questions in the condition stub.

3. Action stub:

It lists statements described all actions that can be taken.

4. Condition stub:

It lists all conditions to be tested for factors necessary for taking a decision.



Decision Tables - Structure

Conditions - *(Condition stub)*	Condition Alternatives – *(Condition Entry)*
Actions – *(Action Stub)*	Action Entries

- Each condition corresponds to a variable, relation or predicate
- Possible values for conditions are listed among the condition alternatives
    - Boolean values (True / False) – Limited Entry Decision Tables
    - Several values – Extended Entry Decision Tables
    - Don’t care value

- Each action is a procedure or operation to perform
- The entries specify whether (or in what order) the action is to be performed

- To express the program logic we can use a limited-entry decision table consisting of 4 areas called the *condition stub*, *condition entry*, *action stub* and the *action entry*:

Condition entry

Condition stub

Action stub

	Rule1	Rule2	Rule3	Rule4
Condition1	Yes	Yes	No	No
Condition2	Yes	X	No	X
Condition3	No	Yes	No	X
Condition4	No	Yes	No	Yes
Action1	Yes	Yes	No	No
Action2	No	No	Yes	No
Action3	No	No	No	Yes

Action Entry

- We can specify *default rules* to indicate the action to be taken when none of the other rules apply.
- When using decision tables as a test tool, default rules and their associated predicates must be explicitly provided.

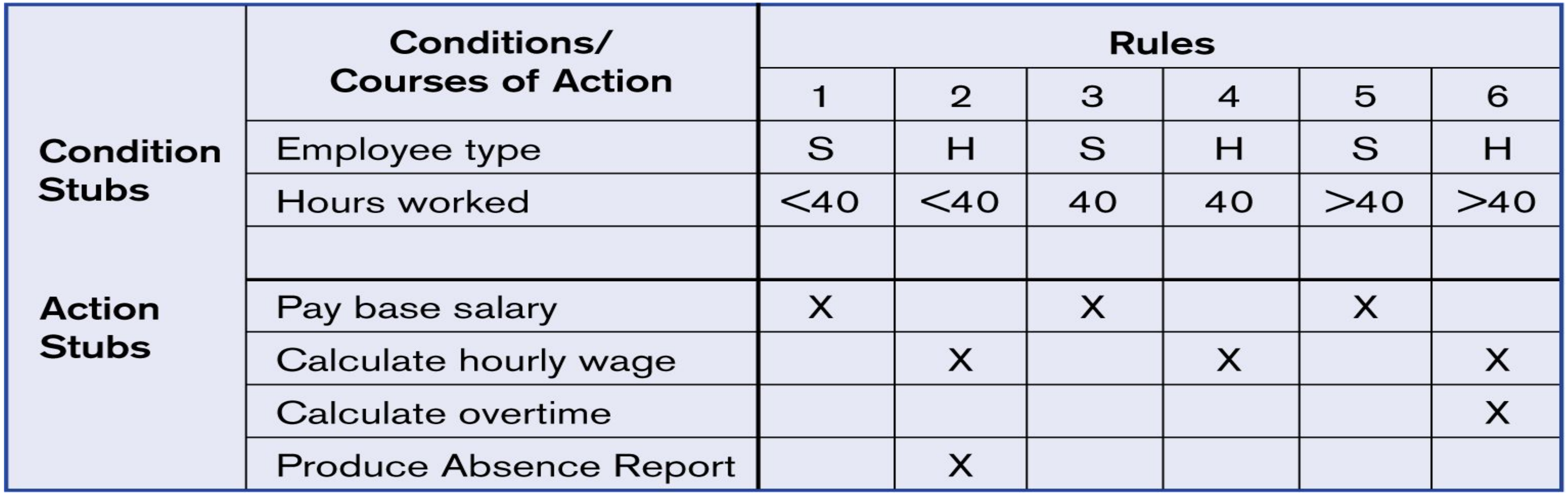
	Rule5	Rule6	Rule7	Rule8
Condition1	X	No	Yes	Yes
Condition2	X	Yes	X	No
Condition3	Yes	X	No	No
Condition4	No	No	Yes	X
Default action	Yes	Yes	Yes	Yes

Decision Table - Example

Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized	Y	N	Y	N	Y	N	Y	N
Actions	Heck the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Printer Troubleshooting

Decision Table Example



	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

Types of Decision Tables

- The decision tables are categorized into two types and these are given below:
- **Limited Entry :** In the limited entry decision tables, the condition entries are restricted to binary values.
- **Extended Entry :** In the extended entry decision table, the condition entries have more than two values. The decision tables use multiple conditions where a condition may have many possibilities instead of only ‘true’ and ‘false’ are known as extended entry decision tables.

Decision Table Development Methodology

1. Determine conditions and values
2. Determine maximum number of rules
3. Determine actions
4. Encode possible rules
5. Encode the appropriate actions for each rule
6. Verify the policy
7. Simplify the rules (reduce if possible the number of columns)

Decision Tables - Issues

- Before using the tables, ensure:
  - rules must be complete
    - every combination of predicate truth values <u>plus</u> default cases are explicit in the decision table
  - rules must be consistent
    - every combination of predicate truth values results in only one action or set of actions

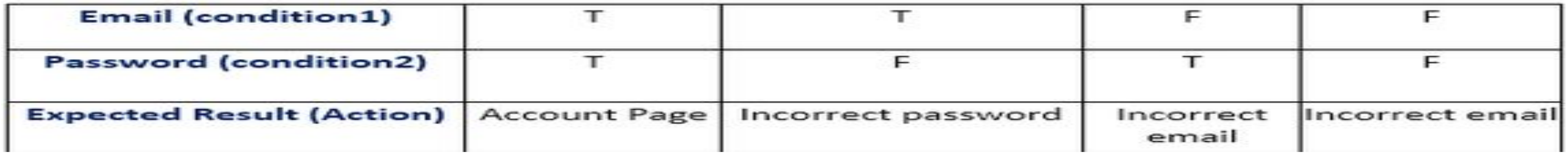
Decision table technique

- **in Black box testing**
- Decision table technique is one of the widely used case design techniques for black box testing. This is a systematic approach where various input combinations and their respective system behavior are captured in a tabular form.
- That's why it is also known as a cause-effect table. This technique is used to pick the test cases in a systematic manner; it saves the testing time and gives good coverage to the testing area of the software application.
- Decision table technique is appropriate for the functions that have a logical relationship between two and more than two inputs.
- This technique is related to the correct combination of inputs and determines the result of various combinations of input. To design the test cases by decision table technique, we need to consider conditions as input and actions as output.

Example

Most of us use an email account, and when you want to use an email account, for this you need to enter the email and its associated password.

- **If both email and password are correctly matched, the user will be directed to the email account's homepage; otherwise, it will come back to the login page with an error message specified with "Incorrect Email" or "Incorrect Password."**
- **Now, let's see how a decision table is created for the login function in which we can log in by using email and password. Both the email and the password are the conditions, and the expected result is action.**



Email (condition1)	T	T	F	F
Password (condition2)	T	F	T	F
Expected Result (Action)	Account Page	Incorrect password	Incorrect email	Incorrect email

- **In the table, there are four conditions or test cases to test the login function. In the first condition if both email and password are correct, then the user should be directed to account's Homepage.**
- **In the second condition if the email is correct, but the password is incorrect then the function should display Incorrect Password. In the third condition if the email is incorrect, but the password is correct, then it should display Incorrect Email.**
- **Now, in fourth and last condition both email and password are incorrect then the function should display Incorrect Email.**
- **In this example, all possible conditions or test cases have been included, and in the same way, the testing team also includes all possible test cases so that upcoming bugs can be cured at testing level.**

Decision Table for the Triangle Problem

Conditions											
C1: a < b+c?	F	T	T	T	T	T	T	T	T	T	T
C2: b < a+c?	-	F	T	T	T	T	T	T	T	T	T
C3: c < a+b?	-	-	F	T	T	T	T	T	T	T	T
C4: a=b?	-	-	-	T	T	T	T	F	F	F	F
C5: a=c?	-	-	-	T	T	F	F	T	T	F	F
C6: b-c?	-	-	-	T	F	T	F	T	F	T	F
Actions											
A1: Not a Triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: Impossible					X	X		X			

Test Cases for the Triangle Problem

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

Guidelines and Observations

- Decision Table testing is most appropriate for programs where
  - there is a lot of decision making
  - there are important logical relationships among input variables
  - There are calculations involving subsets of input variables
  - There are cause and effect relationships between input and output
  - There is complex computation logic (high cyclomatic complexity)
- Decision tables do not scale up very well
- Decision tables can be iteratively refined

Cause Effect Graphing

- **Cause Effect Graphing based technique** is a technique in which a graph is used to represent the situations of combinations of input conditions.
- The graph is then converted to a decision table to obtain the test cases.
- Cause-effect graphing technique is used because boundary value analysis and equivalence class partitioning methods do not consider the combinations of input conditions.
- But since there may be some critical behavior to be tested when some combinations of input conditions are considered, that is why cause-effect graphing technique is used.

Steps used in deriving test cases using this technique are:

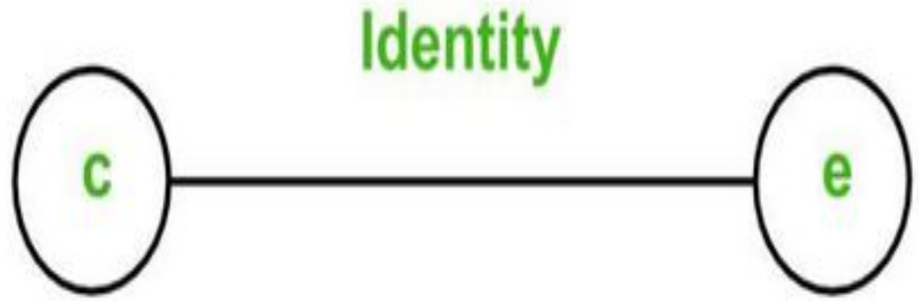
- **Division of specification:**
  Since it is difficult to work with cause-effect graphs of large specifications as they are complex, the specifications are divided into small workable pieces and then converted into cause-effect graphs separately.
- **Identification of cause and effects:**
  This involves identifying the causes(distinct input conditions) and effects(output conditions) in the specification.
- **Transforming the specifications into a cause-effect graph:**
  The causes and effects are linked together using Boolean expressions to obtain a cause-effect graph. Constraints are also added between causes and effects if possible.
- **Conversion into decision table:**
  The cause-effect graph is then converted into a limited entry decision table. If you're not aware of the concept of decision tables, check out this link.
- **Deriving test cases:**
  Each column of the decision-table is converted into a test case.

Basic Notations used in Cause-effect graph:

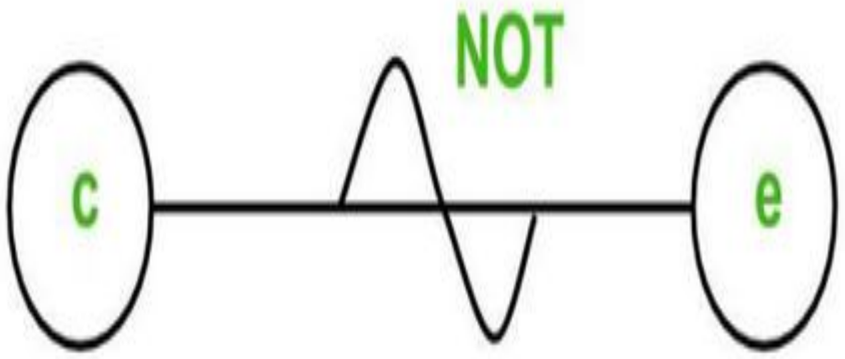
Here **c** represents **cause** and **e** represents **effect**.

The following notations are always **used between a cause and an effect**:

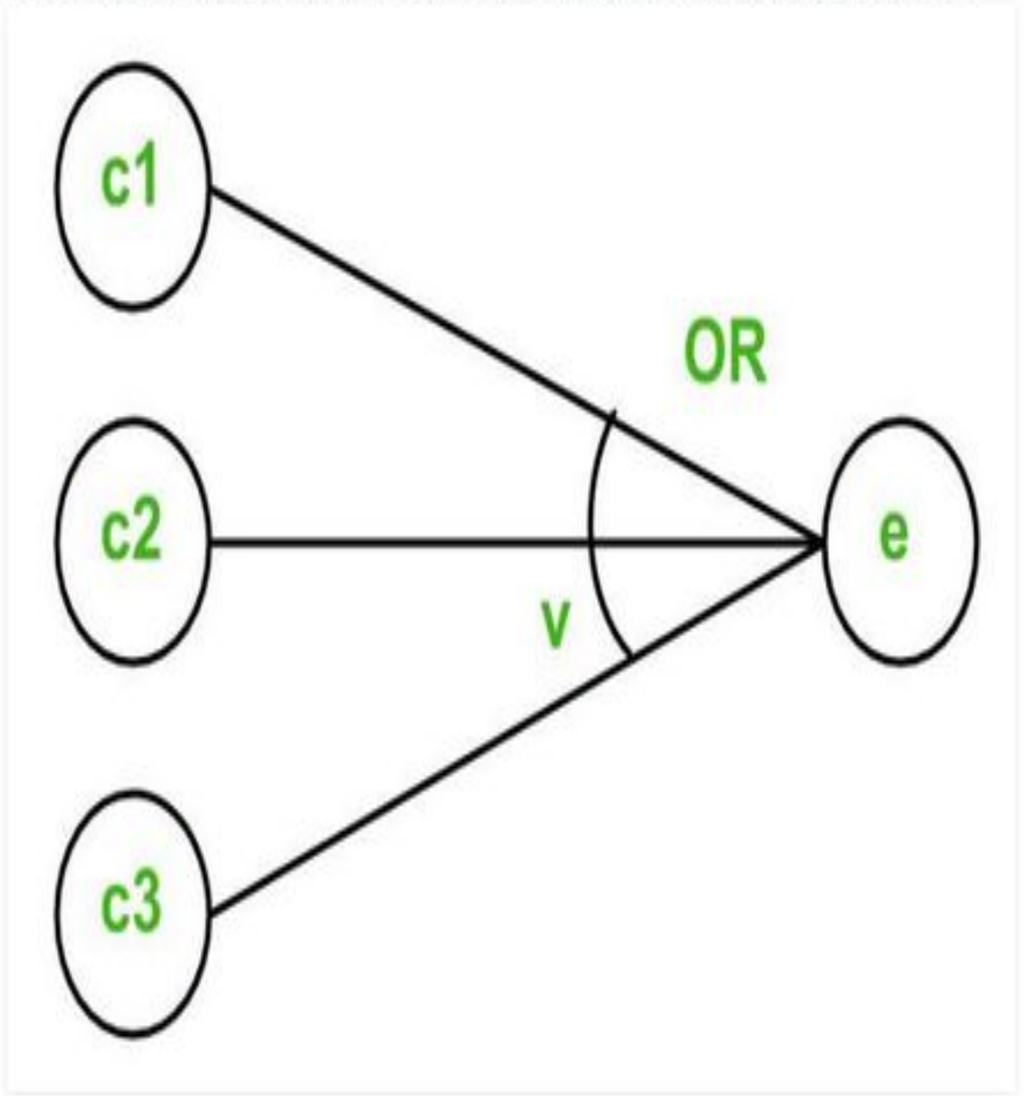
1. **Identity Function:** if c is 1, then e is 1. Else e is 0.



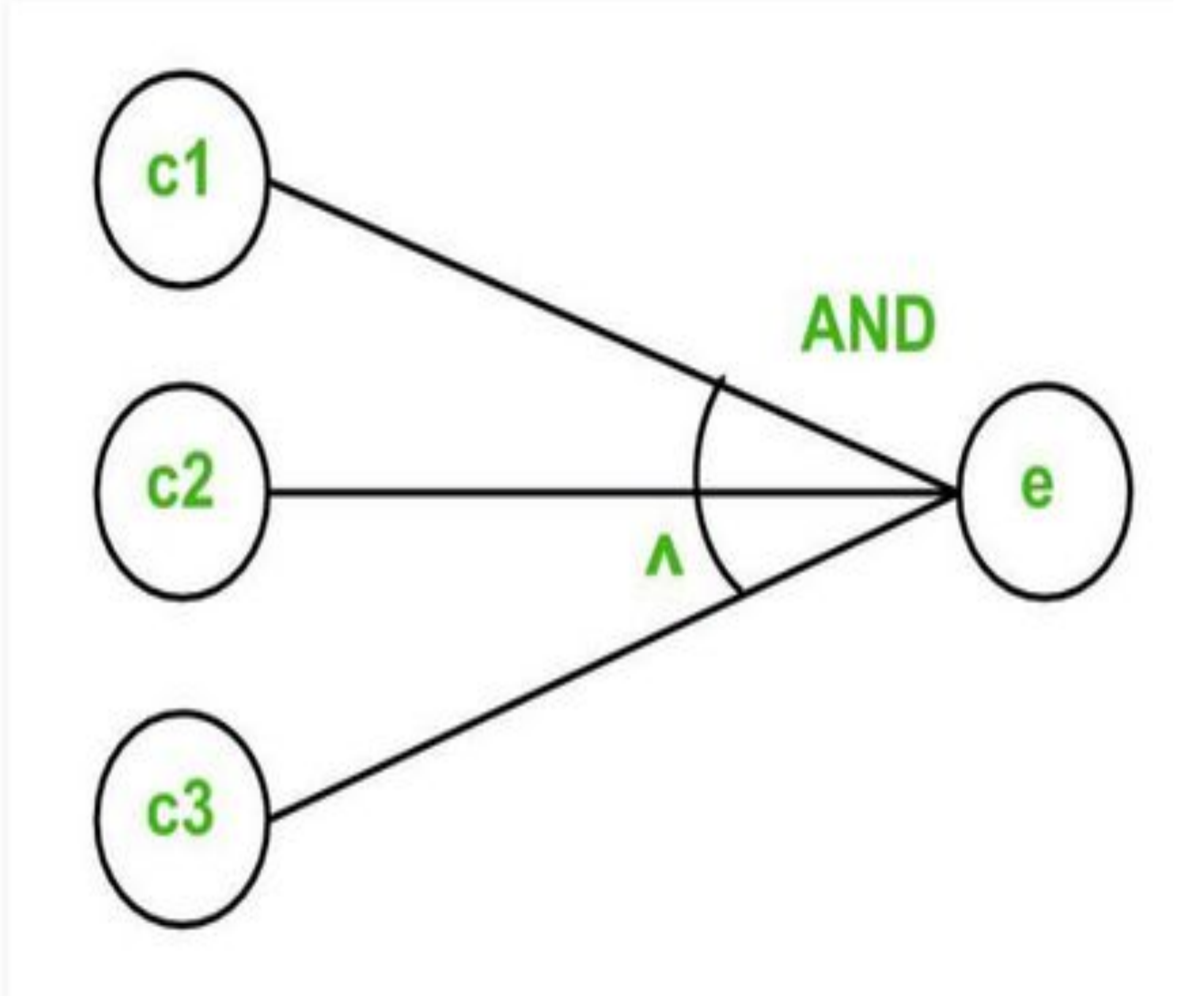
2. **NOT Function:** if c is 1, then e is 0. Else e is 1.



3. **OR Function:** if c1 or c2 or c3 is 1, then e is 1. Else e is 0.



4. **AND Function:** if both c1 and c2 and c3 is 1, then e is 1. Else e is 0.

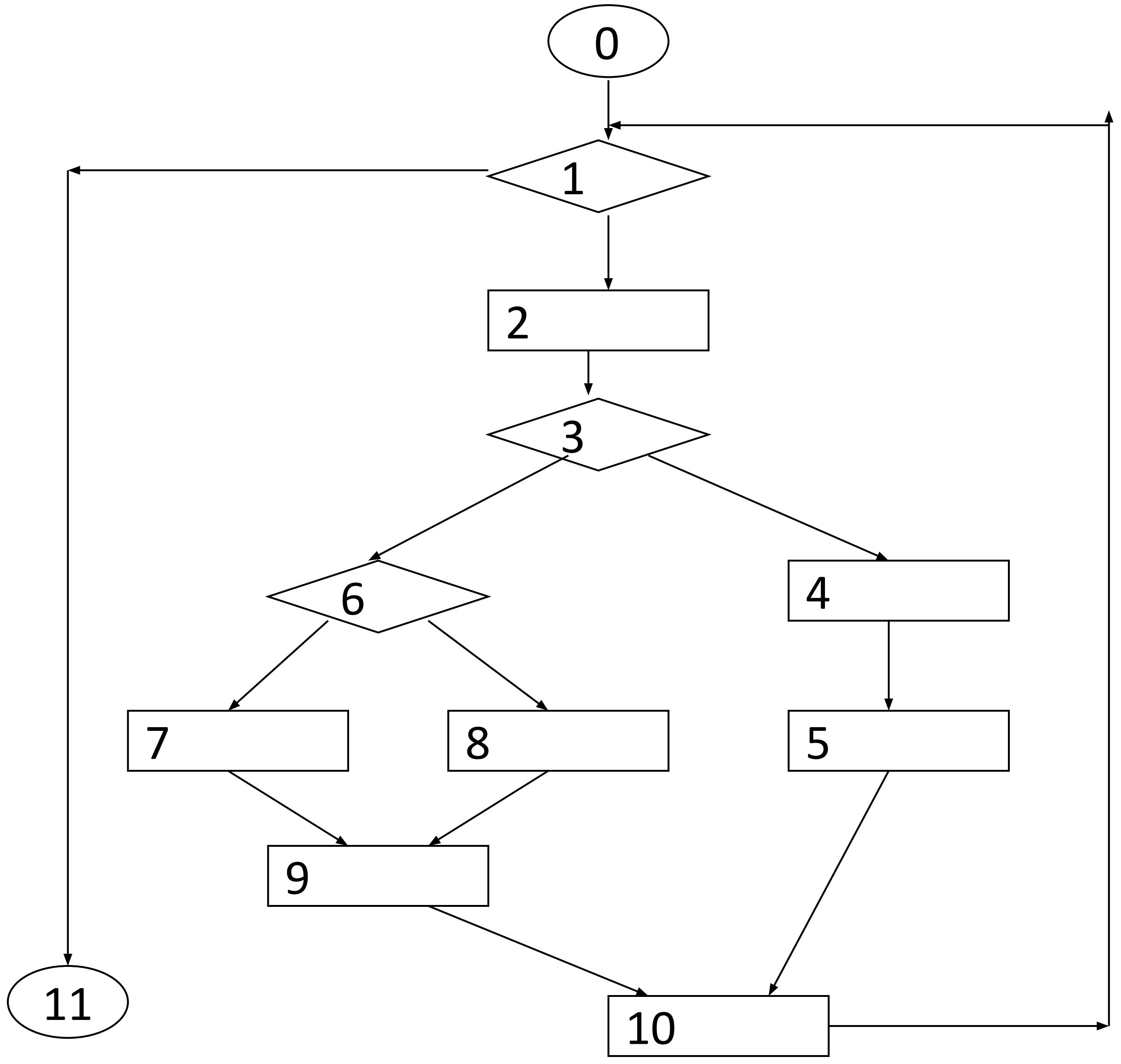


Flow Graph Notation

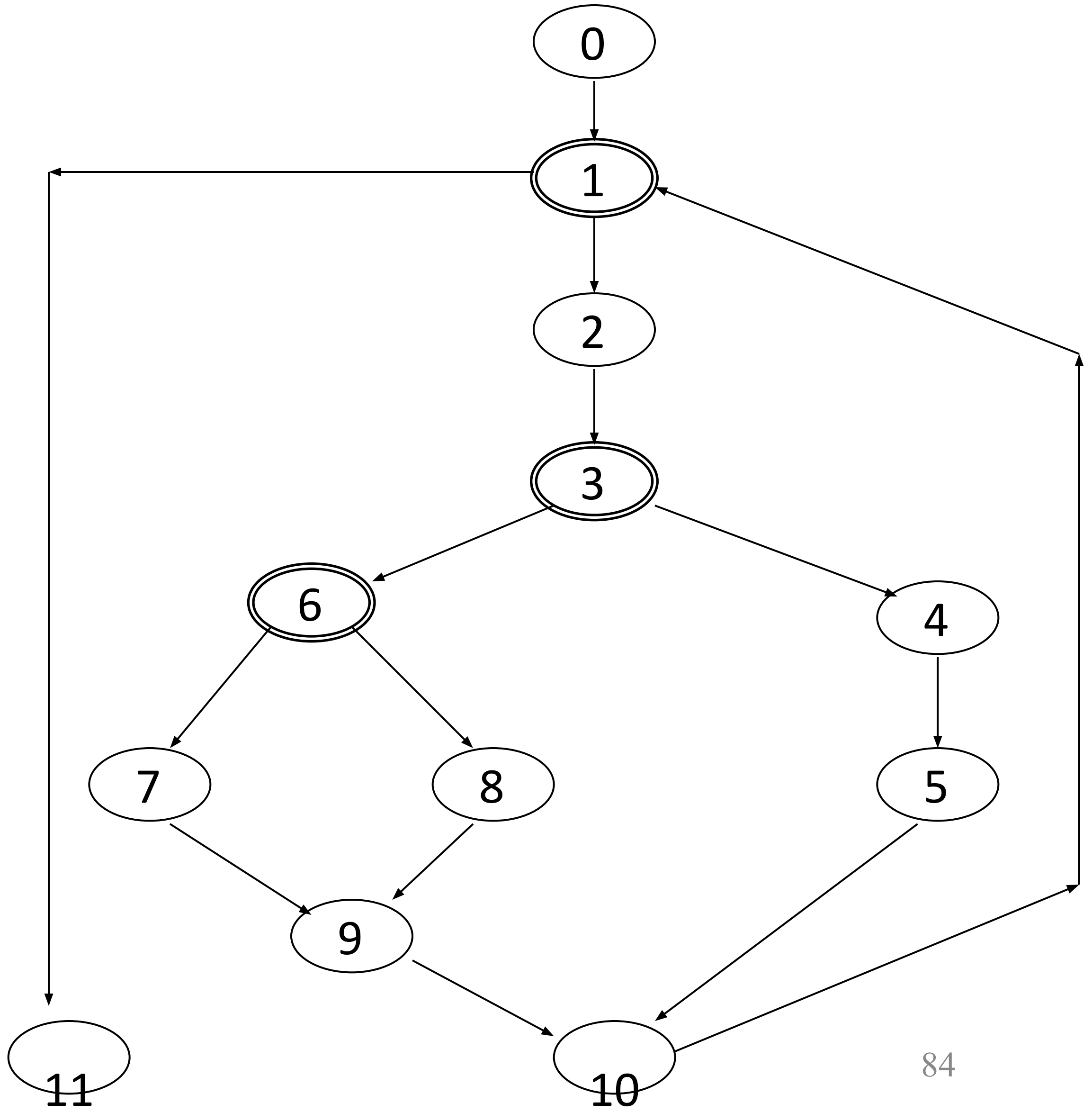
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
  - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
  - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
  - An edge must start and terminate at a node
  - An edge does not intersect or cross over another edge

Flow Graph Example

FLOW CHART



FLOW GRAPH

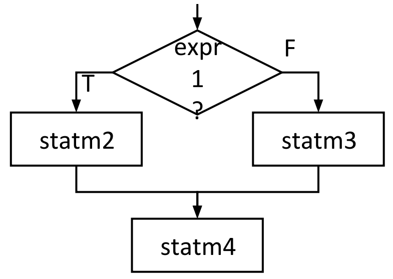


Converting Code to Graph

CODE FLOWCHART GRAPH

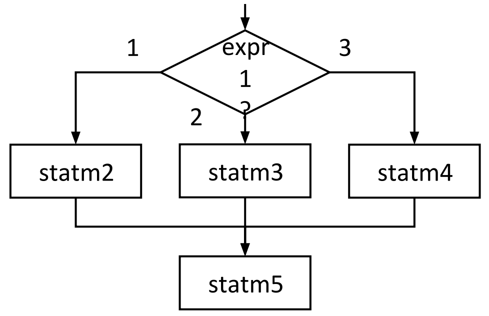
(a)

```
if expression1 then
    statement2
else
    statement3
end if
statement4
```



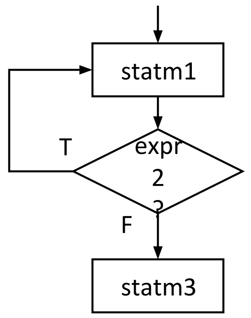
(b)

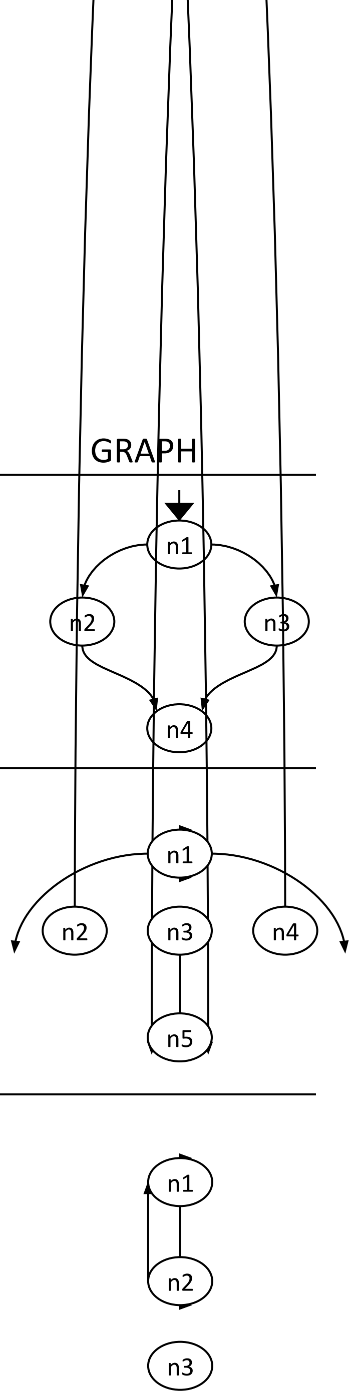
```
switch expr1
    case 1:
        statement2
    case 2:
        statm3
    case 3:
        statm4
end switch
statm5
```



(c)

```
do
    statement1
  while expr2
end do
statement3
```



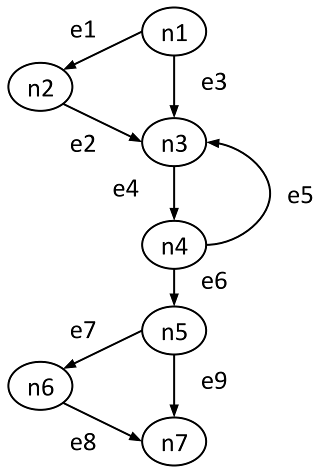


Example Paths

```
if expression1 then
    statement2
end if

do
    statement3
  while expr4
end do

if expression5 then
    statement6
end if
statement7
```



Paths
:
P1 = e1, e2, e4, e6, e7, e8

P2 = e1, e2, e4, e5, e4, e6, e7, e8

P3 = e3, e4, e6, e7, e8, e10

P4 = e6, e7, e8, e10, e3, e4

P5 = e1, e2, e4, e6, e9, e10

P6 = e4, e5

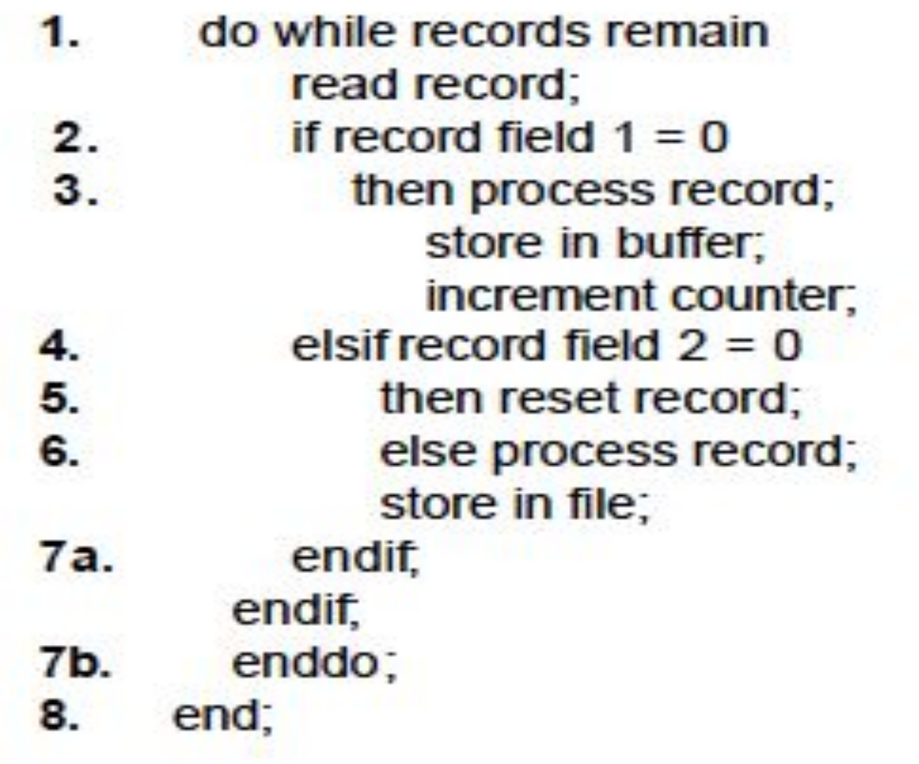
P7 = e3, e4, e6, e9, e10

P8 = e1, e2, e4, e5, e4, e6, e9, e10

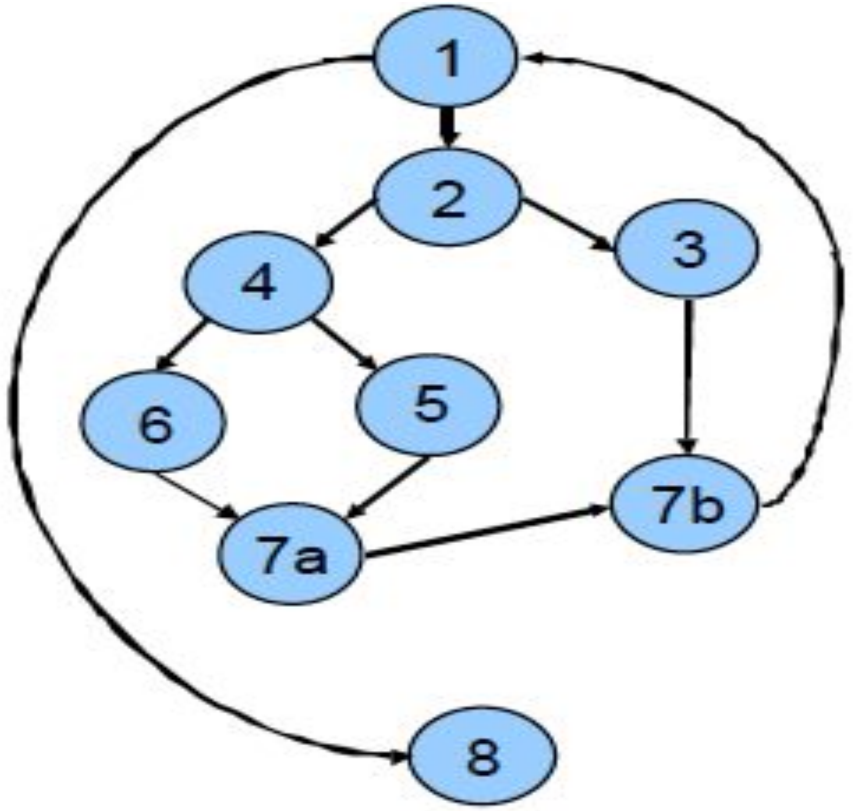
$$\boxed{V(G) = e - n + 2 = 9 - 7 + 2 = 4}$$

Where V(G) is cyclomatic Complexity

Example 1

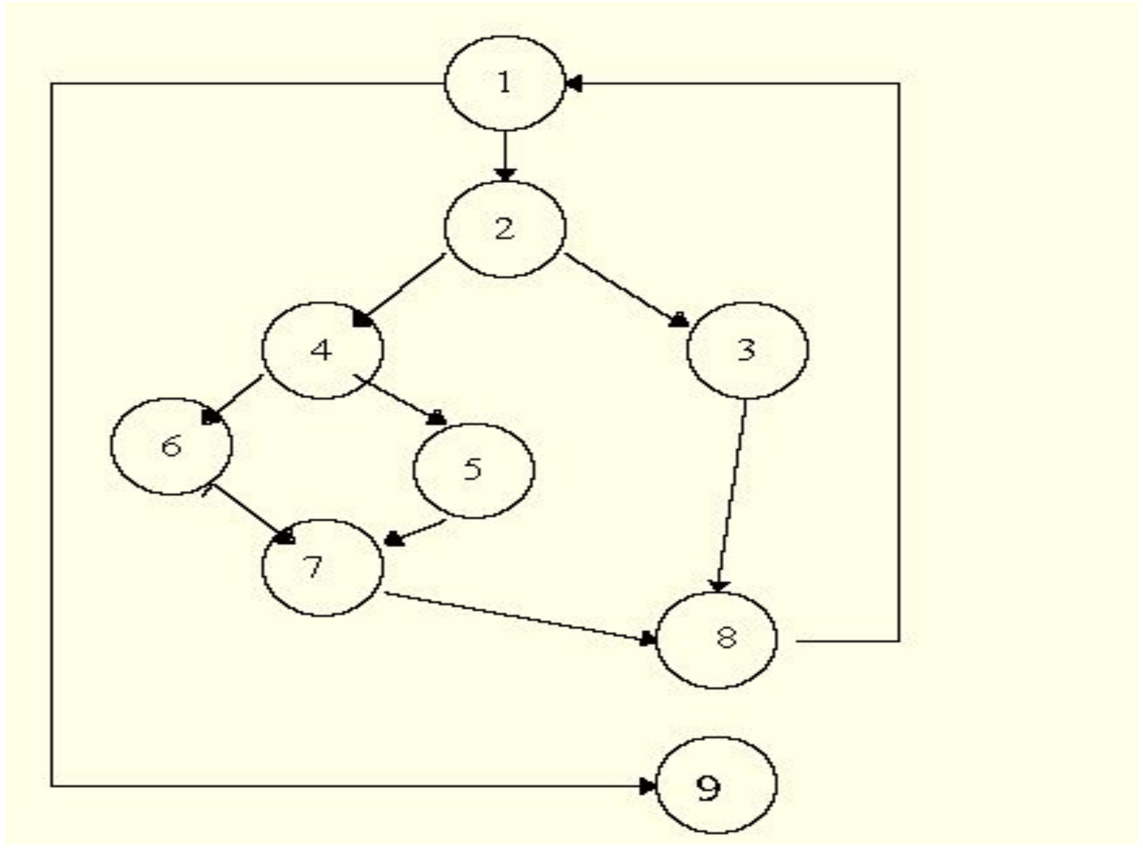


```
1.      do while records remain
            read record;
 2.         if record field 1 = 0
 3.             then process record;
                    store in buffer;
                    increment counter;
4.          elsif record field 2 = 0
5.              then reset record;
6.              else process record;
                store in file;
7a.         endif;
         endif;
7b.      enddo;
8.     end;
```



V=e-n+2=11-9+2=4

```
1:       WHILE NOT EOF LOOP
2:          Read Record;
2:          IF field1 equals 0 THEN
3:             Add field1 to Total
3:             Increment Counter
4:          ELSE
4:              IF field2 equals 0 THEN
5:                  Print Total, Counter
5:                  Reset Counter
6:              ELSE
6:                  Subtract field2 from Total
7:              END IF
8:          END IF
8:          Print "End Record"
9:       END LOOP
9:       Print Counter
```



V=e-n+2=11-9+2=4

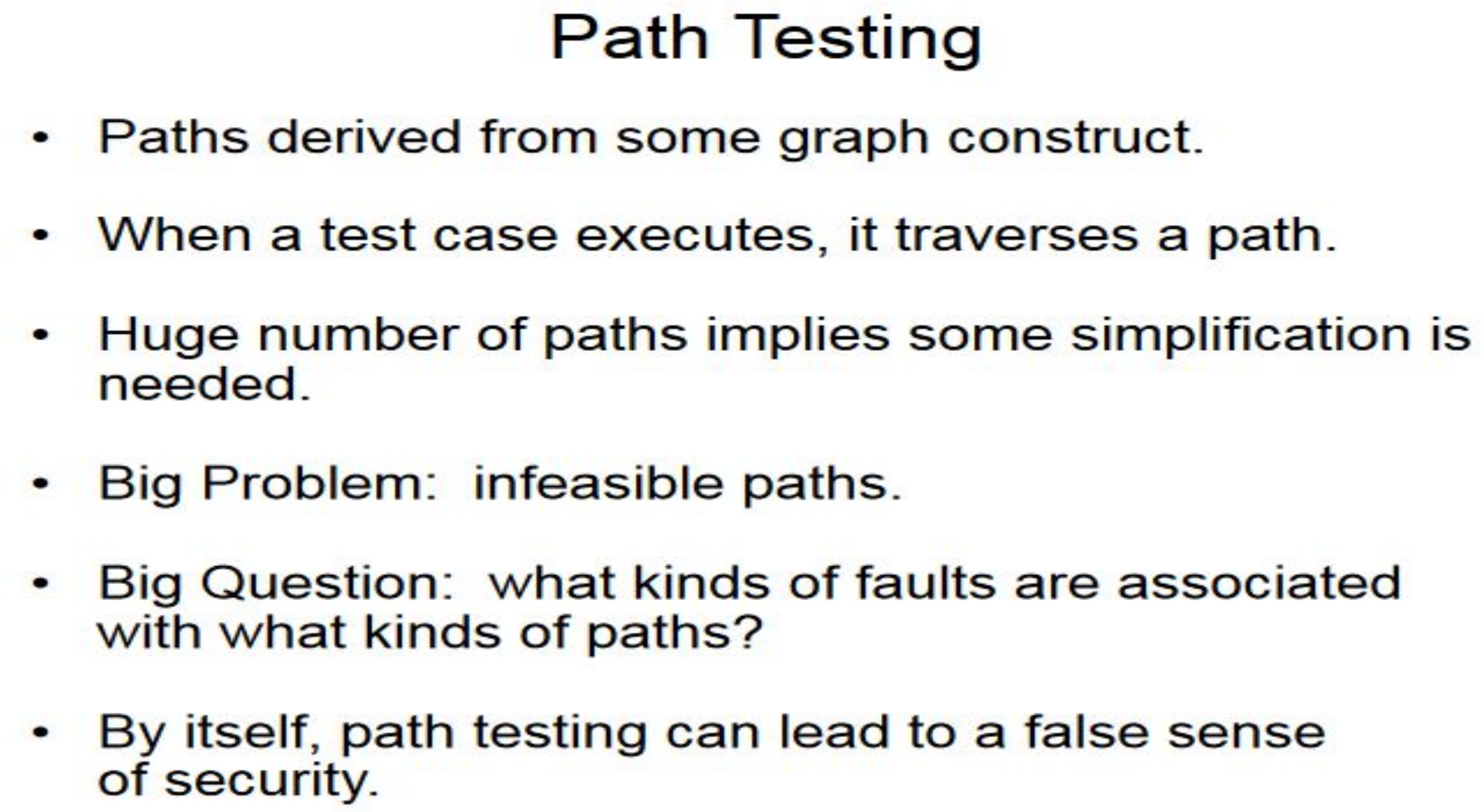
2.2 Path Testing: White Box Testing

- Program Graphs, DD-Paths,
- Test Coverage Metrics,
- Basis Path Testing,
- Guidelines and Observations,
- Data Flow Testing: Define/Use Testing,
- Slice-Based Testing,
- Program Slicing Tools.

Path Testing

What is Path Testing?

- Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path.
- It helps to determine all faults lying within a piece of code.
- This method is designed to execute all or selected path through a computer program.
- Any software program includes, multiple entry and exit points.
- Testing each of these points is a challenging as well as time-consuming.
- In order to reduce the redundant tests and to achieve maximum test coverage, basis path testing is used.



Path Testing

- Paths derived from some graph construct.
- When a test case executes, it traverses a path.
- Huge number of paths implies some simplification is needed.
- Big Problem: infeasible paths.
- Big Question: what kinds of faults are associated with what kinds of paths?
- By itself, path testing can lead to a false sense of security.

Program Graphs

Definition: Given a program written in an imperative programming language, its *program graph* is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a “default” statement fragment.)

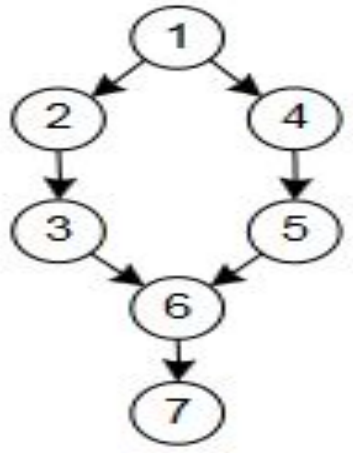
What is a program graph?

- A Control Flow Graph (CFG) or a Program Graph is the graphical representation of control flow or computation during the execution of programs or applications.
- Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit.

Program Graphs of Structured Programming Constructs

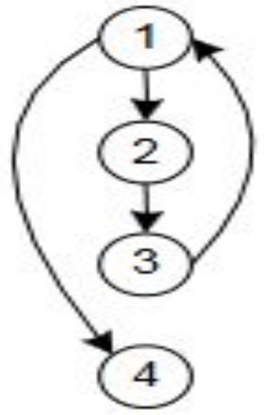
If-Then-Else

```
1   If <condition>
2       Then
3           <then statements>
4       Else
5           <else statements>
6   End If
7   <next statement>
```



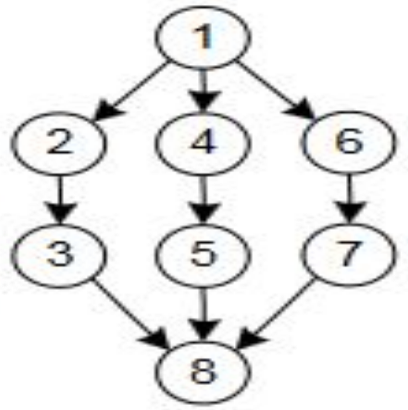
Pre-test Loop

```
1   While <condition>
2           <repeated body>
3   End While
4   <next statement>
```



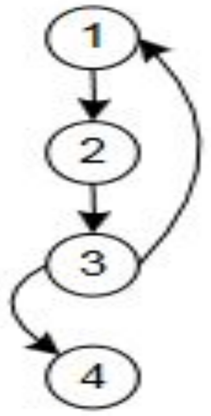
Case/Switch

```
1   Case  n Of 3
2       n=1:
3           <case 1  statements>
4       n=2:
5           <case 2  statements>
6       n=3:
7           <case 3  statements>
8   End Case
```



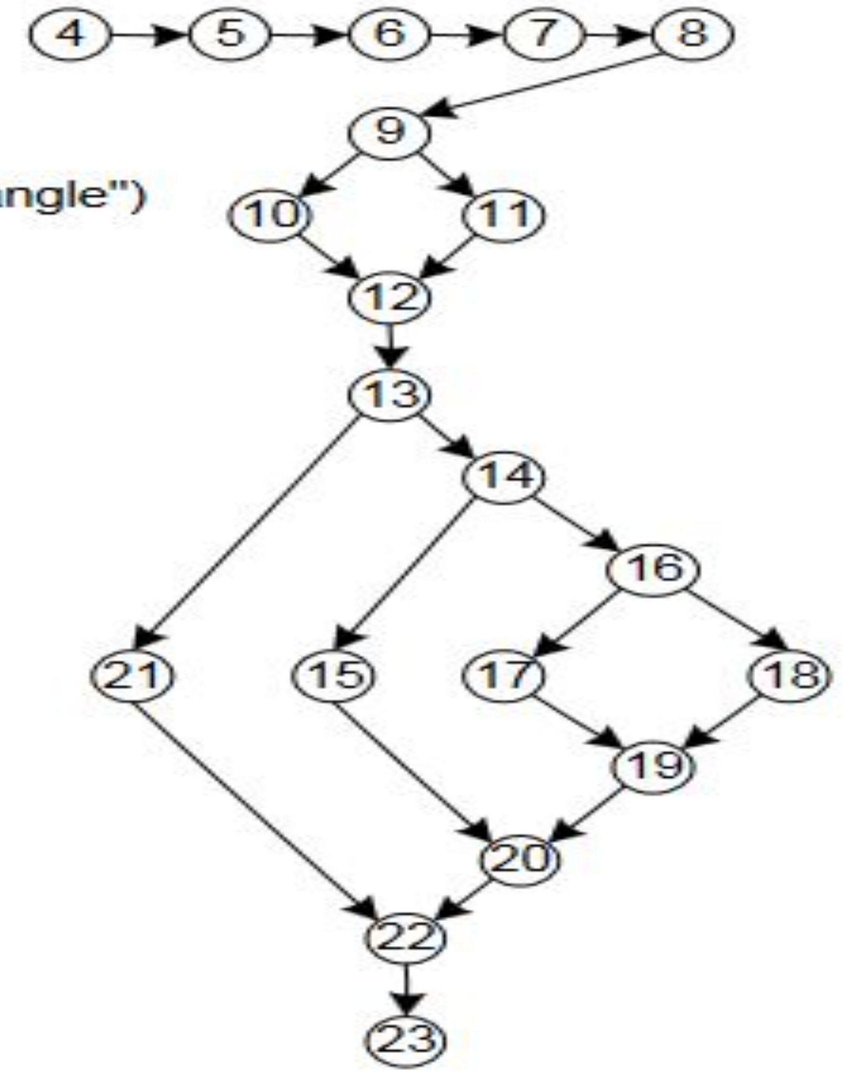
Post-test Loop

```
1   Do
2           <repeated body>
3   Until <condition>
4   <next statement>
```



Sample Program Graph

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATriangle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is ",a)
7  Output("Side B is ",b)
8  Output("Side C is ",c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10     Then IsATriangle = True
11     Else IsATriangle = False
12  EndIf
13  If IsATriangle
14     Then  If (a = b) AND (b = c)
15             Then Output ("Equilateral")
16             Else    If (a≠b) AND (a≠c) AND (b≠c)
17                         Then  Output ("Scalene")
18                         Else  Output ("Isosceles")
19                     EndIf
20             EndIf
21     Else    Output("Not a Triangle")
22  EndIf
23  End triangle2
```



Control Flow Graph (CFG) : Program Graph

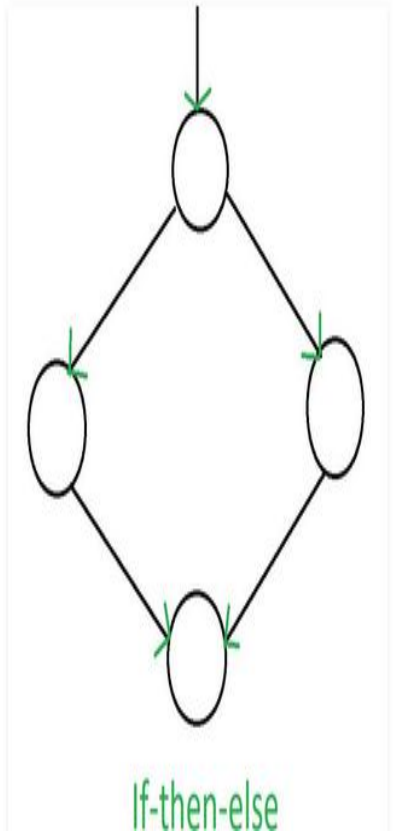
- A **Control Flow Graph (CFG)** is the graphical representation of control flow or computation during the execution of programs or applications.
- Control flow graphs are mostly used in static analysis as well as compiler applications, as they can accurately represent the flow inside of a program unit.
- The control flow graph was originally developed by *Frances E. Allen*.
- **Characteristics of Control Flow Graph:**
- Control flow graph is process oriented.
- Control flow graph shows all the paths that can be traversed during a program execution.
- Control flow graph is a directed graph.
- Edges in CFG portray control flow paths and the nodes in CFG portray basic blocks.

- There exist 2 designated blocks in Control Flow Graph:
- **Entry Block:**
  Entry block allows the control to enter into the control flow graph.
- **Exit Block:**
  Control flow leaves through the exit block.
- Hence, the control flow graph is comprised of all the building blocks involved in a flow diagram such as the start node, end node and flows between the nodes.

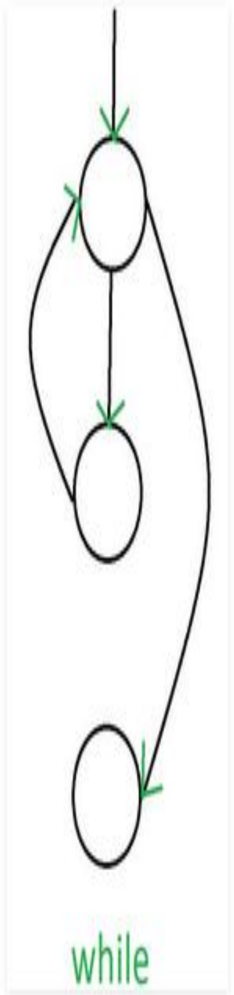
General Control Flow Graphs:

Control Flow Graph is represented differently for all statements and loops. Following images describe it:

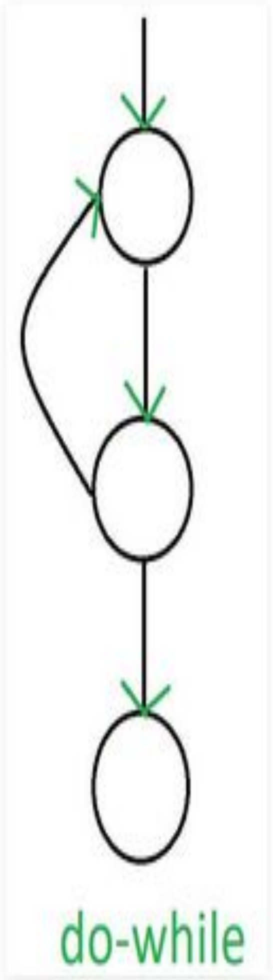
1. **If-else:**



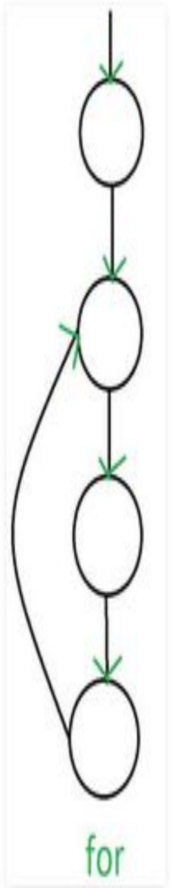
2. **while:**

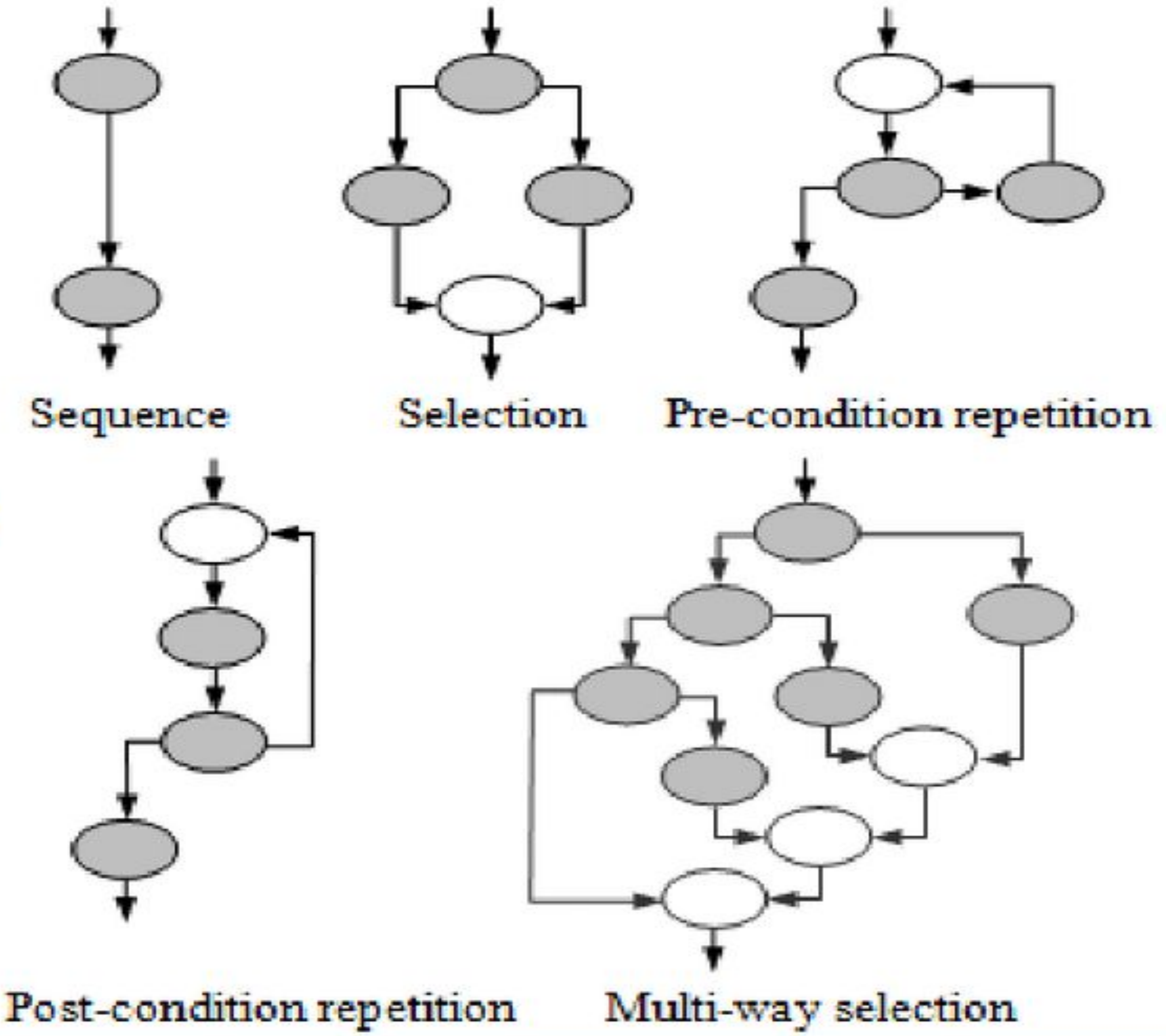


3. **do-while:**



4. **for:**

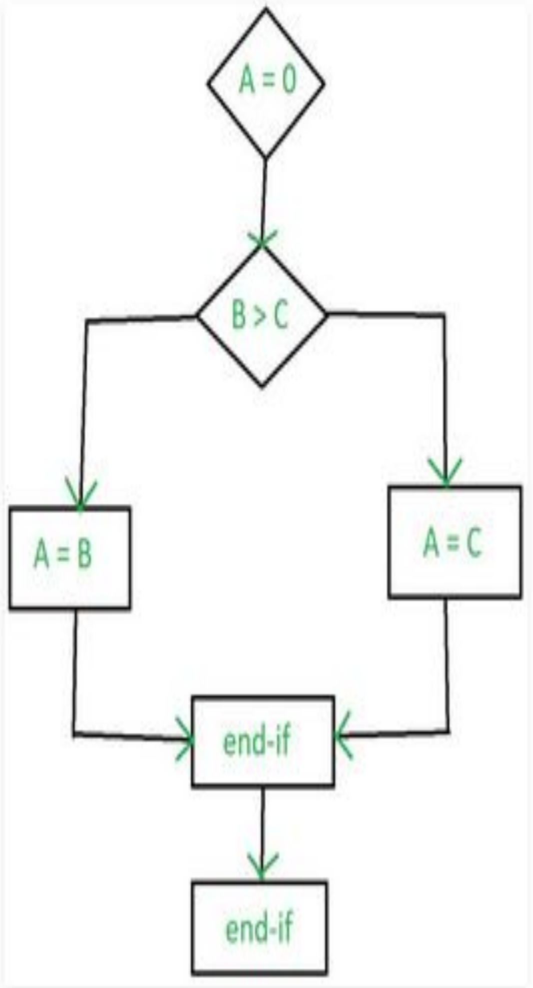



Sequence
Selection
Pre-condition repetition
Post-condition repetition
Multi-way selection

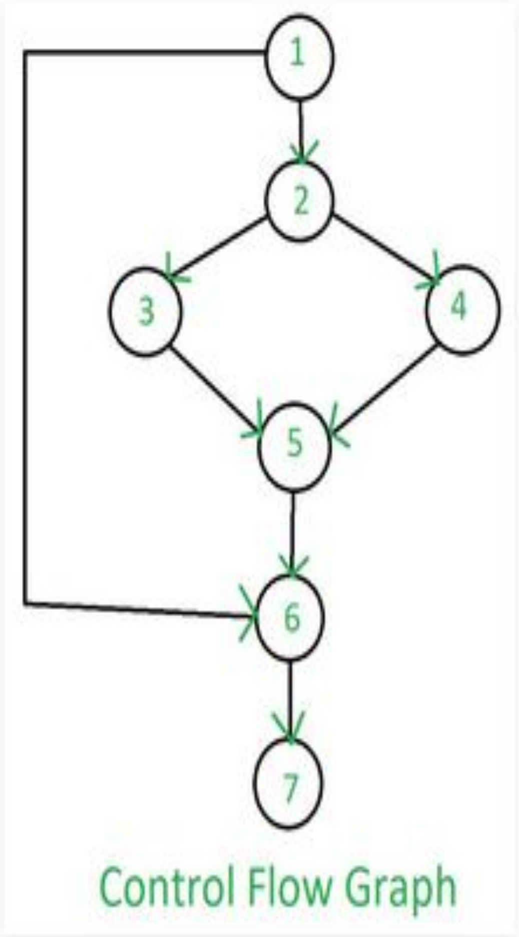
Example:

```
if  A = 10 then
  if B > C
    A = B
  else A = C
  endif
  endif
print A, B, C
```

Flowchart of above example will be:

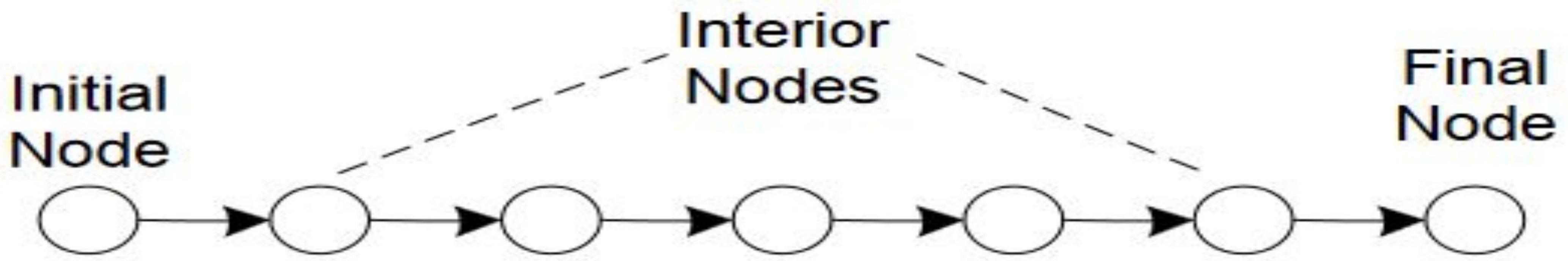


Control Flow Graph of above example will be:



DD-Paths

- Originally defined by E. F. Miller (1977?)
- "DD" is short for "decision to decision"
- Original definition was for early (second generation) programming languages
- Similar to a "chain" in a directed graph
- Bases of interesting test coverage metrics



DD-Paths

A *DD-Path* (decision-to-decision) is a chain in a program graph such that

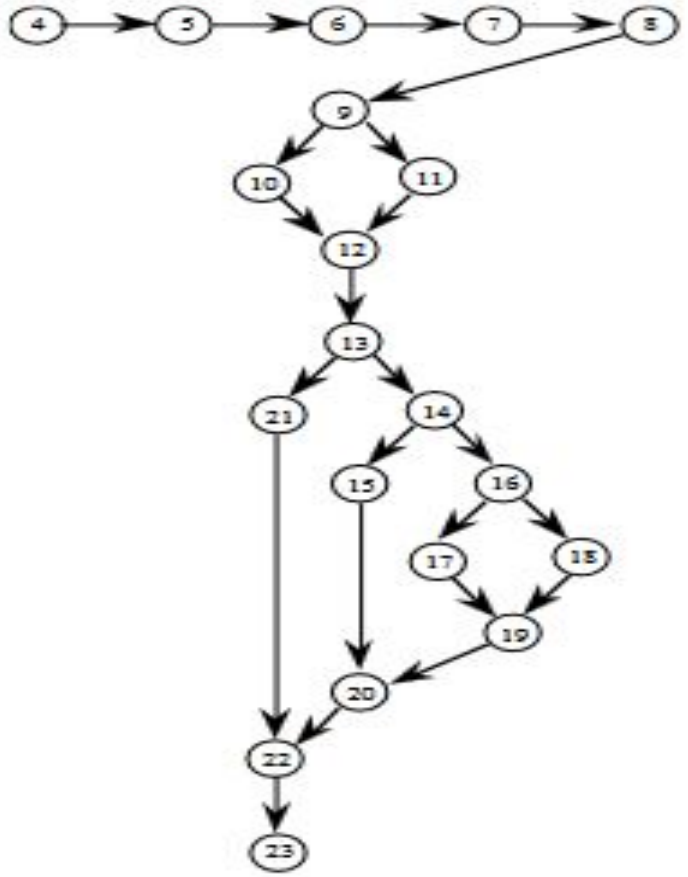
Case 1: it consists of a single node with indeg = 0,
Case 2: it consists of a single node with outdeg = 0,
Case 3: it consists of a single node with indeg ≥ 2 or
 outdeg ≥ 2,
Case 4: it consists of a single node with indeg = 1 and
 outdeg = 1,
Case 5: it is a maximal chain of length ≥ 1.

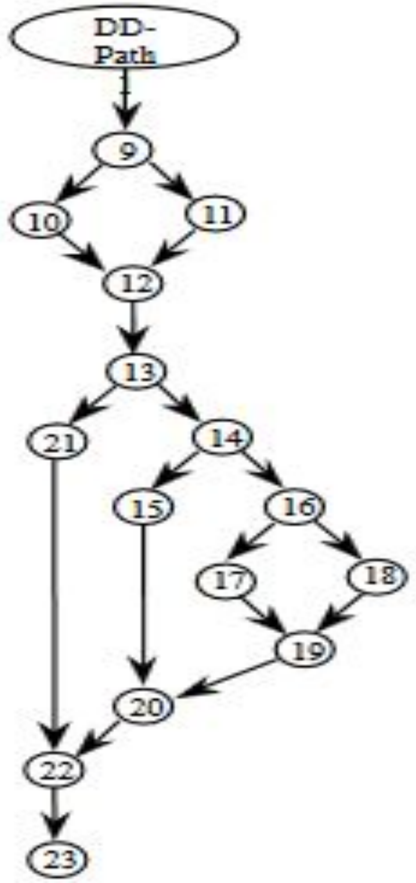
DD-Path Graph

Given a program written in an imperative language, its *DD-Path graph* is the directed graph in which nodes are DD-Paths of its program graph, and edges represent control flow between successor DD-Paths.

- a form of condensation graph
- 2-connected components are collapsed into an individual node
- single node DD-Paths (corresponding to Cases 1 - 4) preserve the convention that a statement fragment is in exactly one DD-Path

DD-Path Graph of the Triangle Program
(not much compression because this example is control intensive, with little sequential code.)





Code-Based Test Coverage Metrics

- Used to evaluate a given set of test cases
- Often required by
  - contract
  - U.S. Department of Defense
  - company-specific standards
- Elegant way to deal with the gaps and redundancies that are unavoidable with specification-based test cases.
- BUT
  - coverage at some defined level may be misleading
  - coverage tools are needed

Code-Based Test Coverage Metrics

(E. F. Miller, 1977 dissertation)

- C_0: Every statement
- C_1: Every DD-Path
- C_{1p}: Every predicate outcome
- C_2: C_1 coverage + loop coverage
- C_d: C_1 coverage +every pair of dependent DD-Paths
- C_{MCC}: Multiple condition coverage
- C_{ik}: Every program path that contains up to k repetitions of a loop (usually k = 2)
- C_{stat}: "Statistically significant" fraction of paths
- C_∞: All possible execution paths

Test Coverage Metrics from Program Graphs

- Every node
- Every edge
- Every chain
- Every path

- How do these compare with Miller’ s coverage metrics?

Basis Path Testing in Software Engineering

- **Basis Path Testing** in software engineering is a White Box Testing method in which test cases are defined based on flows or logical paths that can be taken through the program.
- The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.
- In software engineering, Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases.
- It is a hybrid method of branch testing and path testing methods.

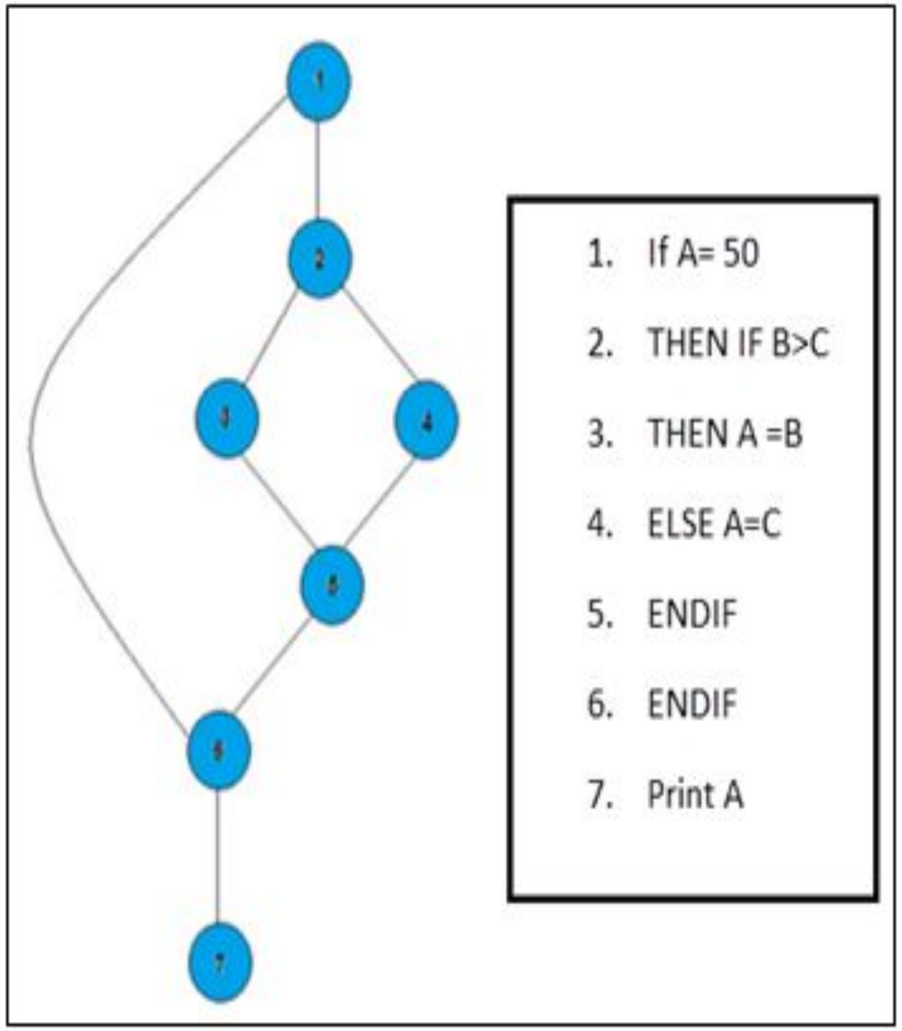
Example

In this there are few conditional statements that are executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

Path 1: 1,2,3,5,6, 7

Path 2: 1,2,4,5,6, 7

Path 3: 1, 6, 7



```
1. If A= 50
2. THEN IF B>C
3. THEN A =B
4. ELSE A=C
5. ENDIF
6. ENDIF
7. Print A
```

Steps for Basis Path testing

1. The basic steps involved in basis path testing include
2. Draw a control graph (to determine different program paths)
3. Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
4. Find a basis set of paths
5. Generate test cases to exercise each path

- **Advantages of Basic Path Testing**

1. It helps to reduce the redundant tests
2. It focuses attention on program logic
3. It helps facilitates analytical versus arbitrary case design
4. Test cases which exercise basis set will execute every statement in a program at least once

Basis Path Testing

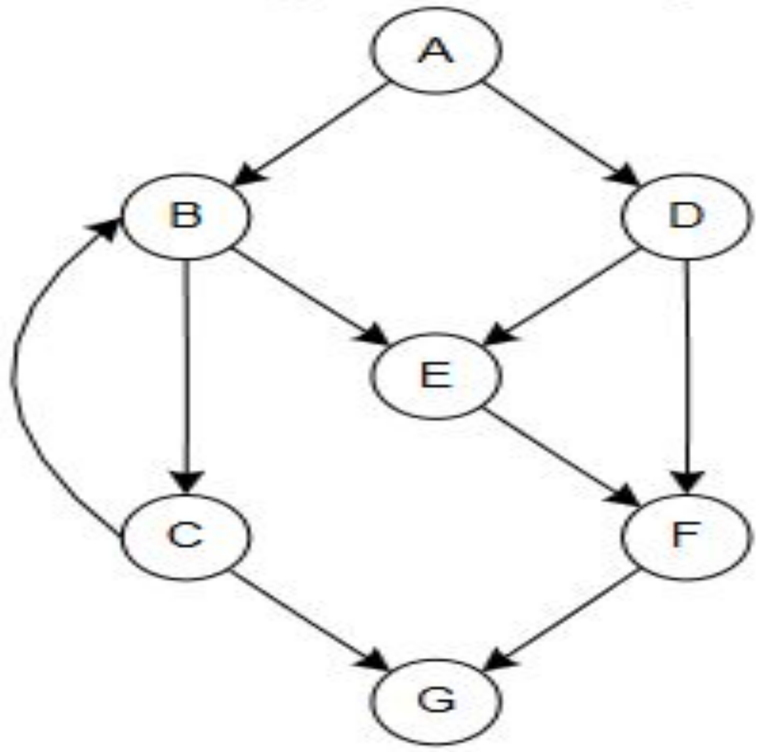
- Proposed by Thomas McCabe in 1982
- Math background
  - a Vector Space has a set of independent vectors called basis vectors
  - every element in a vector space can be expressed as a linear combination of the basis vectors
- Example: Euclidean 3-space has three basis vectors
  - (1, 0, 0) in the x direction
  - (0, 1, 0) in the y direction
  - (0, 0, 1) in the z direction
- The Hope: If a program graph can be thought of as a vector space, there should be a set of basis vectors. Testing them tests many other paths.

(McCabe) Basis Path Testing

- in math, a basis "spans" an entire space, such that everything in the space can be derived from the basis elements.
- the cyclomatic number of a strongly connected directed graph is the number of linearly independent cycles.
- given a program graph, we can always add an edge from the sink node to the source node to create a strongly connected graph. (assuming single entry, single exit)
- computing $V(G) = e - n + p$ from the modified program graph yields the number of independent paths that must be tested.
- since all other program execution paths are linear combinations of the basis path, it is necessary to test the basis paths. (Some say this is sufficient; but that is problematic.)
- the next few slides follow McCabe's original example.

McCabe's Example

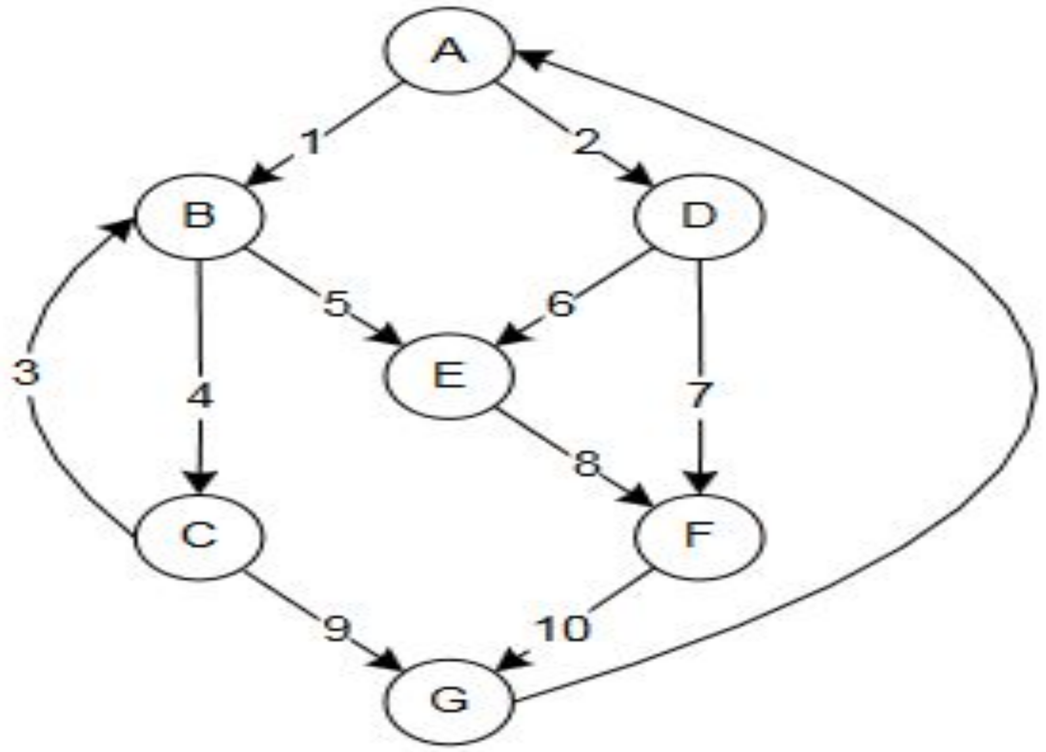
McCabe's Original Graph



$V(G) = 10 - 7 + 2(1)$

$= 5$

Derived, Strongly Connected Graph



$V(G) = 11 - 7 + 1$

$= 5$

McCabe's Baseline Method

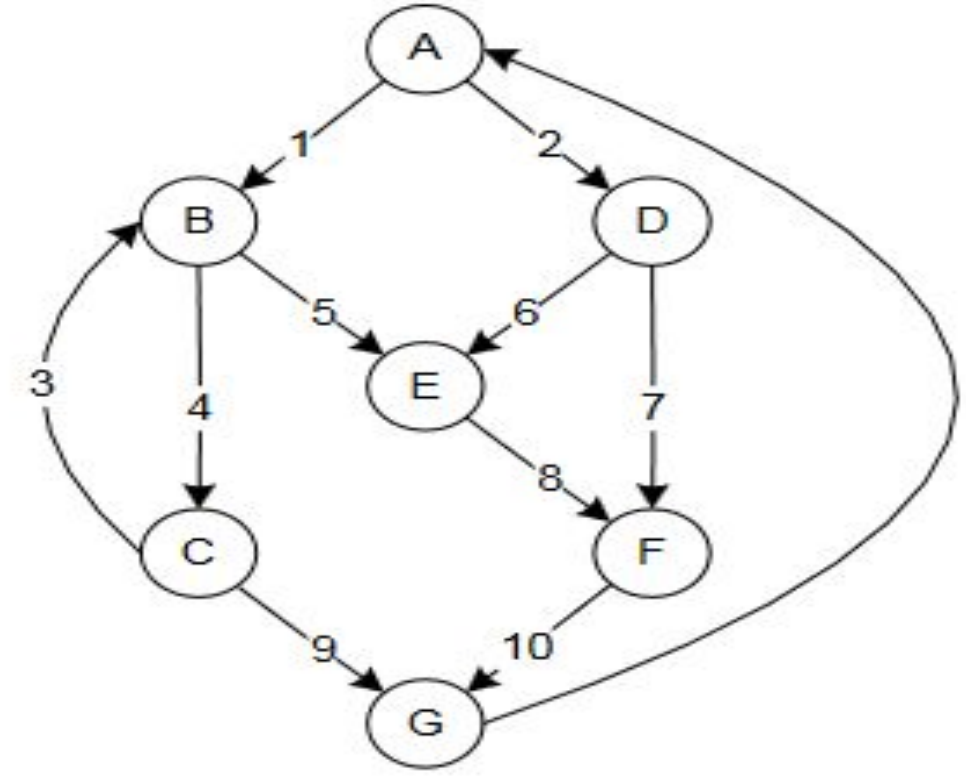
- Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)
- To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision (take another alternative) and continue as much of the baseline as possible.
- Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.
- If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

Following this algorithm, we get basis paths for McCabe's example.

McCabe's Example (with numbered edges)

Resulting basis paths

First baseline path
p1: A, B, C, G
Flip decision at C
p2: A, B, C, B, C, G
Flip decision at B
p3: A, B, E, F, G
Flip decision at A
p4: A, D, E, F, G
Flip decision at D
p5: A, D, F, G



Path/Edge Incidence

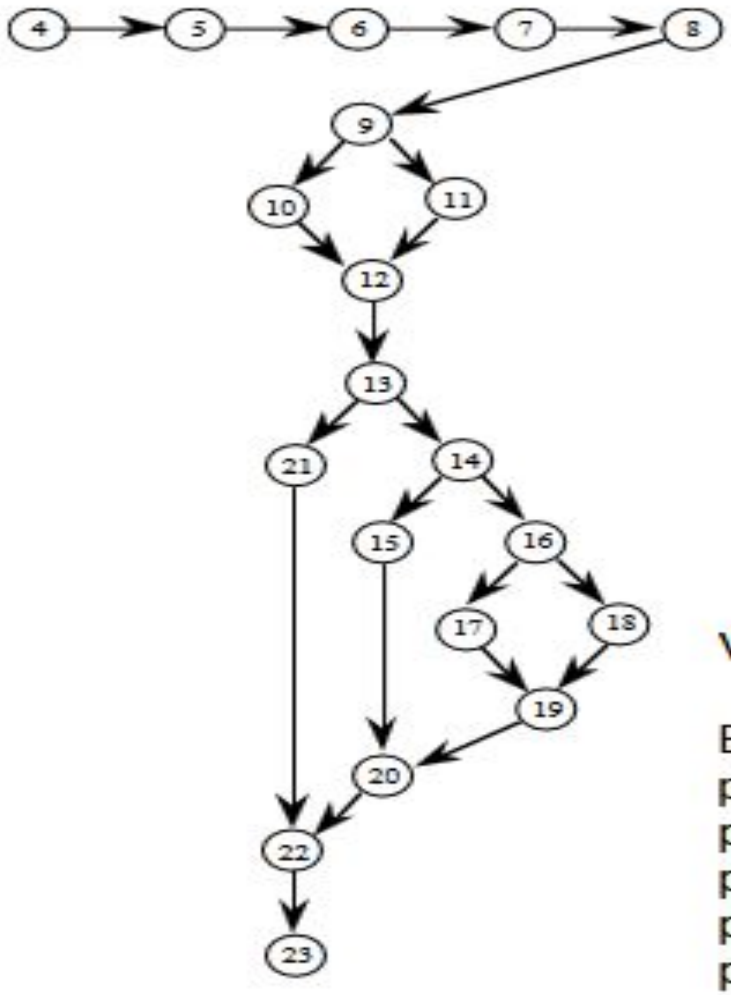
Path / Edges	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	**1**	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	**1**	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	**1**	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	**1**	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Sample paths as linear combinations of basis paths
ex1 = p2 + p3 - p1
ex2 = 2p2 - p1

Problems with Basis Paths

- What is the significance of a path as a linear combination of basis paths?
- What do the coefficients mean? What does a minus sign mean?
- In the path ex2 = 2p2 - p1 should a tester run path p2 twice, and then not do path p1 the next time? This is theory run amok.
- Is there any guarantee that basis paths are feasible?
- Is there any guarantee that basis paths will exercise interesting dependencies?

McCabe Basis Paths in the Triangle Program



There are 8 topologically possible **paths**.
4 are feasible, and 4 are infeasible.

Exercise: Is every basis path feasible?

V(G) = 23 - 20 + 2(1) = 5

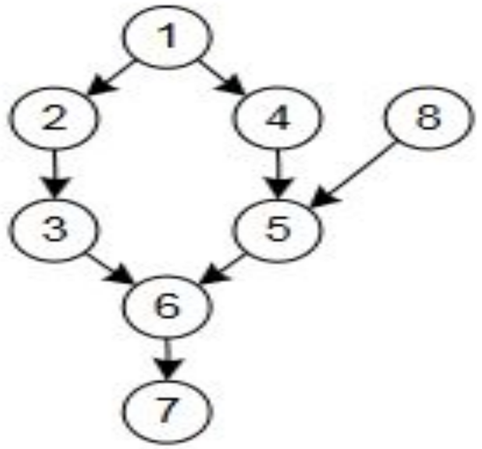
Basis Path Set B1
p1: 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 18, 19, 20, 22, 23 (mainline)
p2: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 18, 19, 20, 22, 23 (flipped at 9)
p3: 4, 5, 6, 7, 8, 9, 11, 12, 13, 21, 22, 23 (flipped at 13)
p4: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 20, 22, 23 (flipped at 14)
p5: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 19, 20, 22, 23 (flipped at 16)

Essential Complexity

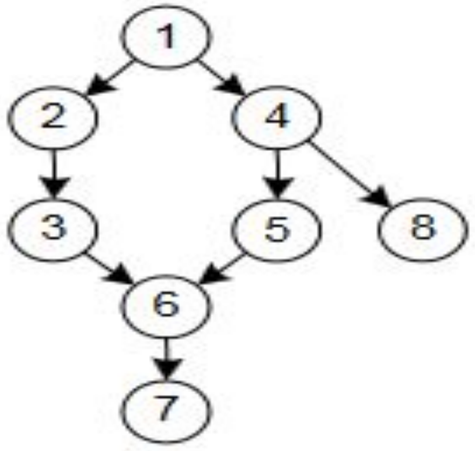
- McCabe’s notion of Essential Complexity deals with the extent to which a program violates the precepts of Structured Programming.
- To find Essential Complexity of a program graph,
  - Identify a group of source statements that corresponds to one of the basic Structured Programming constructs.
  - Condense that group of statements into a separate node (with a new name)
  - Continue until no more Structured Programming constructs can be found.
  - The Essential Complexityof the original program is the cyclomatic complexity of the resulting program graph.
- The essential complexity of a Structured Program is 1.
- Violations of the precepts of Structured Programming increase the essential complexity.

Violations of Structured Programming Precepts

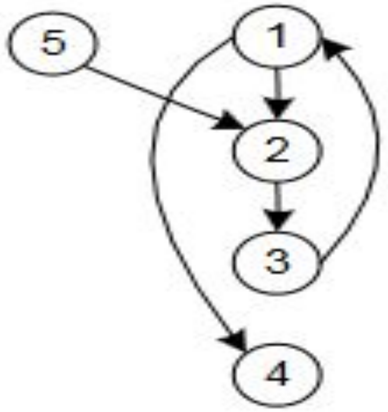
Branching into a decision



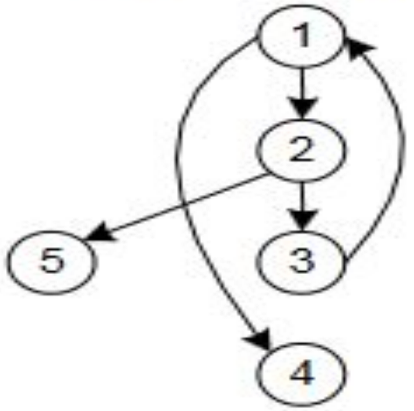
Branching out of a decision



Branching into a loop



Branching out of a loop



Cons and Pros of McCabe’ s Work

- Issues
  - Linear combinations of execution paths are counter-intuitive. What does 2p2 – p1 really mean?
  - How does the baseline method guarantee feasible basis paths?
  - Given a set of feasible basis paths, is this a sufficient test?
- Advantages
  - McCabe's approach does address both gaps and redundancies.
  - Essential complexity leads to better programming practices.
  - McCabe proved that violations of the structured programming constructs increase cyclomatic complexity, and violations cannot occur singly.

Data Flow Testing

- Data flow testing is a **family of test strategies based on selecting paths through the program's control flow** in order to explore sequences of events related to the status of variables or data objects.
- Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.
- It is a type of structural testing.
- It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.
  It is concerned with:
- Statements where variables receive values,
- Statements where these values are used or referenced.

Dataflow Testing

- Often confused with "dataflow diagrams“.
- Main concern: places in a program where data values are defined and used.
- Static (compile time) and dynamic (execution time) versions.
- Static: Define/Reference Anomalies on a variable that
  - is defined but never used (referenced)
  - is used but never defined
  - is defined more than once
- Starting point is a program, P, with program graph G(P), and the set V of variables in program P.
- "Interesting" data flows are then tested as separate functions.

The General Idea

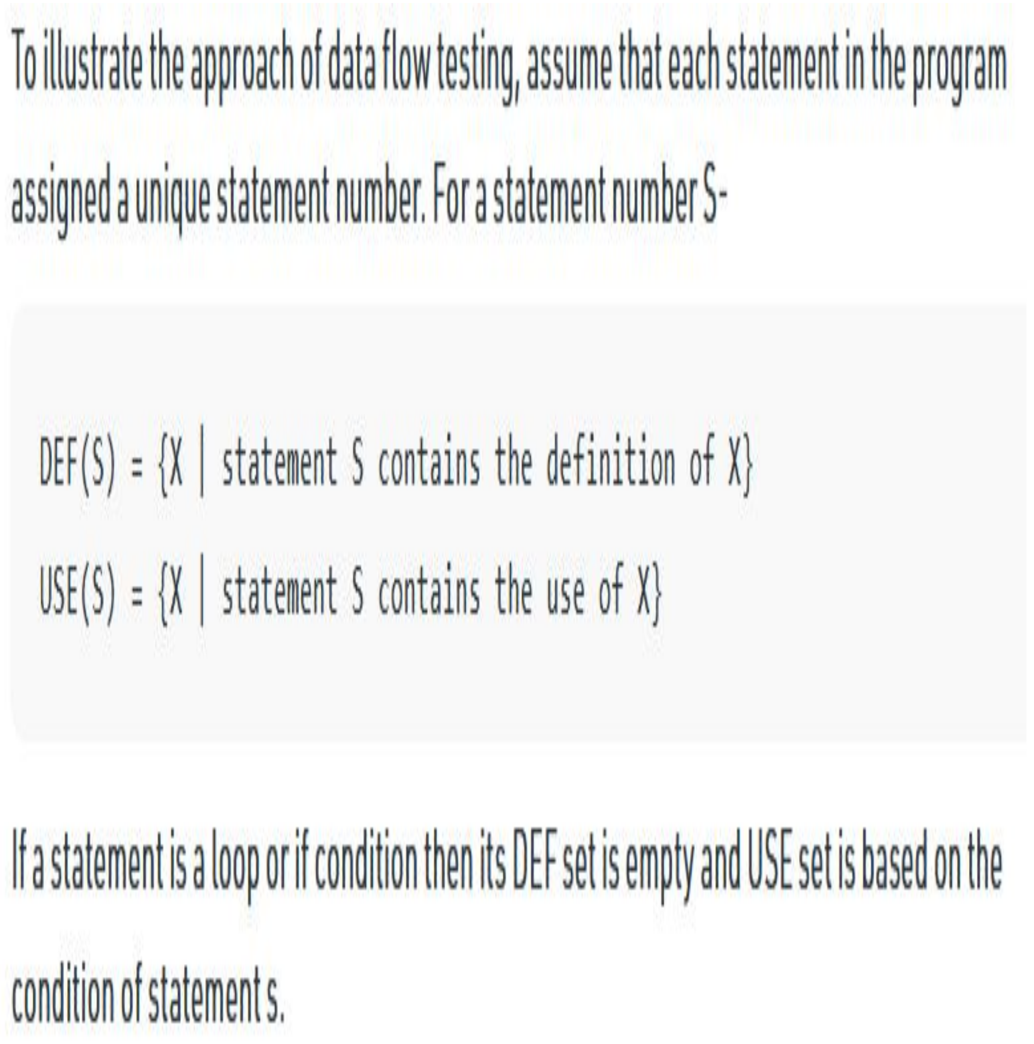
- Data flow testing can be performed at two conceptual levels.
  - Static data flow testing
  - Dynamic data flow testing
- Static data flow testing
  - Identify potential defects, commonly known as **data flow anomaly.**
  - Analyze source code.
  - Do not execute code.
- Dynamic data flow testing
  - Involves actual program execution.
  - Bears similarity with control flow testing.
    - Identify paths to execute them.
    - Paths are identified based on **data flow testing criteria**.

Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.
Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:

A variable is defined but not used or referenced,

A variable is used but never defined,

A variable is defined twice before it is used


To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S-

```
DEF(S) = {X | statement S contains the definition of X}

USE(S) = {X | statement S contains the use of X}
```

If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.

- **Advantages of Data Flow Testing:**
  Data Flow Testing is used to find the following issues-
- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is use,
- Deallocating a variable before it is used.

- **Disadvantages of Data Flow Testing**
- Time consuming and costly process
- Requires knowledge of programming languages

Data Flow Graph

- 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 edge parameter and each nonlocal variable used in the program.
  - The exit node has an undefinition of each local variable.

Data Flow Terms

- **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.
  - Example: The c-use of variable tv in node 9 (Figure 5.4) is a global c-use.

- **Definition clear path**: A path (i – n_1 – … n_m – j), m ≥ 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),

  if x has been neither defined nor undefined in nodes n_1 – … n_m.
  - Example: (2 – 3 – 4 – 6 – 3 – 4 – 6 – 3 – 4 – 5) is a def-clear path w.r.t. tv in Fig. 5.4.
  - Example: (2 – 3 – 4 – 5) and (2 – 3 – 4 – 6) are def-clear paths w.r.t. variable tv from node 2 to 5 and from node 2 to 6, respectively, in Fig. 5.4.

Data Flow Terms

- **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.
  - Example: The c-use of variable tv in node 9 (Figure 5.4) is a global c-use.

- **Definition clear path**: A path ($i – n_1 – … n_m – j$), $m \geq 0$, 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),

  if x has been neither defined nor undefined in nodes $n_1 – … n_m$.
  - Example: (2 – 3 – 4 – 6 – 3 – 4 – 6 – 3 – 4 – 5) is a def-clear path w.r.t. tv in Fig. 5.4.
  - Example: (2 – 3 – 4 – 5) and (2 – 3 – 4 – 6) are def-clear paths w.r.t. variable tv from node 2 to 5 and from node 2 to 6, respectively, in Fig. 5.4.

Data Flow Terms

- **Global definition**: A node i has a global definition of variable x if node i has a definition of x and there is a def-clear path w.r.t. x from node i to some

    node containing a global c-use, or

    edge containing a p-use of

  variable x.

- **Simple path**: A simple path is a path in which all nodes, except possibly the first and the last, are distinct.
  - Example: Paths (2 – 3 – 4 – 5) and (3 – 4 – 6 – 3) are simple paths.
- **Loop-free paths**: A loop-free path is a path in which all nodes are distinct.
- **Complete path**: A complete path is a path from the entry node to the exit node.

Data Flow Terms

- **Du-path**: A path ($n_1 – n_2 – … – n_j – n_k$) is a du-path path w.r.t. variable x if node n_1 has a global definition of x and <u>either</u>
    - node n_k has a global c-use of x and ($n_1 – n_2 – … – n_j – n_k$) is a def-clear simple path w.r.t. x, <u>or</u>
    - Edge (n_j, n_k) has a p-use of x and ($n_1 – n_2 – … – n_j – n_k$) is a def-clear, loop-free path w.r.t. x.

  - Example: Considering the global definition and global c-use of variable tv in nodes 2 and 5, respectively, (2 – 3 – 4 – 5) is a du-path.
  - Example: Considering the global definition and p-use of variable tv in nodes 2 and on edge (7, 9), respectively, (2 – 3 – 7 – 9) is a du-path.

Data Flow Terms

- **Du-path**: A path ($n_1 – n_2 – \ldots – n_j – n_k$) is a du-path path w.r.t. variable x if node n_1 has a global definition of x and <u>either</u>
    - node n_k has a global c-use of x and ($n_1 – n_2 – \ldots – n_j – n_k$) is a def-clear simple path w.r.t. x, <u>or</u>
    - Edge (n_j, n_k) has a p-use of x and ($n_1 – n_2 – \ldots – n_j – n_k$) is a def-clear, loop-free path w.r.t. x.

- Example: Considering the global definition and global c-use of variable tv in nodes 2 and 5, respectively, (2 – 3 – 4 – 5) is a du-path.
- Example: Considering the global definition and p-use of variable tv in nodes 2 and on edge (7, 9), respectively, (2 – 3 – 7 – 9) is a du-path.

Data Flow Testing Criteria

- Seven data flow testing criteria
  - All-defs
  - All-c-uses
  - All-p-uses
  - All-p-uses/some-c-uses
  - All-c-uses/some-p-uses
  - All-uses
  - All-du-paths

Data Flow Testing Criteria

- **All-defs**
  - For ***each variable*** x and ***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.
  - Example (**partial**): Consider tv with its global definition in node 2. Variable tv has a global c-use in node 5, and there is a def-clear path (2 – 3 – 4 – 5) from node 2 to node 5. Choose a complete path (1 – $\underline{2 – 3 – 4 – 5}$ – 6 – 3 – 7 – 9 – 10) that includes the def-clear path (2 – 3 – 4 – 5) to satisfy the all-defs criterion.

Data Flow Testing Criteria

- **All-c-uses**
  - For ***each variable*** x and ***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.
  - Example (**partial**): Consider variable ti, which has a global definition in 2 and a global c-use in node 4. From node 2, the def-clear path to 4 is (2 – 3 – 4). One may choose the complete path (1 – <u>2 – 3 – 4</u> – 6 – 3 – 7 – 8 – 10). (There three other complete paths.)
- All-p-uses
  - For ***each variable*** x and ***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 (j, k).
  - Example (**partial**): Consider variable tv, which has a global definition in 2 and p-uses on edges (7, 8) and (7, 9). From node 2, there are def-clear paths to (7, 8) and (7, 9), namely (2 – 3 – 7 – 8) and (2 – 3 – 7 – 9). The two complete paths are: (1 – <u>2 – 3 – 7 – 8</u> – 10) and (1 – <u>2 – 3 – 7 – 9</u> – 10).

Data Flow Testing Criteria

- All-p-uses/some-c-uses
  - This criterion is identical to the all-p-uses criterion **except** when a variable x has no p-use. If x has no p-use, then this criterion reduces to the some-c-uses criterion.
  - Some-c-uses: For ***each variable*** x and ***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 ***some*** nodes j such that there is a global c-use of x in j.

  - Example (**partial**): Consider variable i, which has a global definition in 2. There is no p-use of i. Corresponding to the global definition of I in 2, there is a global c-use of I in 6. The def-clear path from node 2 to 6 is (2 – 3 – 4 – 5 – 6). A complete path that includes the above def-clear path is (1 – <u>2 – 3 – 4 – 5 – 6</u> – 7 – 9 – 10).

Data Flow Testing Criteria

- **All-c-uses/some-p-uses**
  - This criterion is identical to the all-c-uses criterion **except** when a variable x has no c-use. If x has no global c-use, then this criterion reduces to the some-p-uses criterion.
  - Some-p-uses: For ***each variable*** x and ***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 ***some*** edges (j, k) such that there is a p-use of x on (j, k).
- **All-uses**: This criterion produces a set of paths due to the **all-p-uses** criterion **and** the **all-c-uses** criterion.
- **All-du-paths**: For each variable x and for each node i, such that x has a global definition in node i, select complete paths which include **all du-paths** from node i
  - To all nodes j such that there is a global **c-use** of x in j, and
  - To all edges (j, k) such that there is a **p-use** of x on (j, k).

Definitions

- Given a program, P, with a set V of variables in P, and the program graph G(P), Node n ∈ G(P) is
  - a *defining node of the variable* v ∈ V, written as DEF(v, n), iff the value of the variable v is defined at the statement fragment corresponding to node n.
  - *a usage node of the variable* v ∈ V, written as USE(v, n), iff the value of the variable v is used at the statement fragment corresponding to node n.
- A usage node USE(v, n) is a *predicate use* (denoted as P-use) iff the statement n is a predicate statement; otherwise, USE(v, n) is a *computation use* (denoted C-use).

More Definitions

- A *definition-use path with respect to a variable* v (denoted du-path) is a path in the set of all paths in P, PATHS(P), such that for some v ∈ V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the path.

- A *definition-clear path with respect to a variable* v (denoted dc-path) is a definition-use path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the path is a defining node of v.

Dataflow Testing Strategies

- Dataflow testing is indicated in
  - Computation-intensive applications
  - “long” programs
  - Programs with many variables
- A definition-clear du-path represents a small function that can be tested by itself.
- If a du-path is not definition-clear, it should be tested for each defining node.

Slice Testing Definitions

Starting point is a program, P, with program graph G(P), and the set V of variables in program P. Nodes in the program graph are numbered and correspond to statement fragments.

- Definition: The *slice on the variable set V at statement fragment n*, written S(V, n), is the set of node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n.
- This is actually a "backward slice".
- Exercise: define a "forward slice".

Fine Points

- "prior to" is the dynamic part of the definition.
- "contribute" is best understood by extending the Define and Use concepts:
  - P-use: used in a predicate (decision)
  - C-use: used in computation
  - O-use: used for output
  - L-use: used for location (pointers, subscripts)
  - I-use: iteration (internal counters, loop indices)
  - I-def: defined by input
  - A-def: defined by assignment

Fine Points (continued)

- usually, the set V of variables consists of just one element.
- can choose to define a slice as a compilable set of statement fragments -- this extends the meaning of "contribute"
- because slices are sets, we can develop a lattice based on the subset relationship.

Slice Based testing

- Slicing or program slicing is a technique used in software testing which takes a slice or **a group of program statements in the program for testing particular test conditions or cases** and that may affect a value at a particular point of interest.
- What is slicing and dicing in software testing?
- When one thinks of slicing, **filtering is done to focus on a particular attribute**, dicing on the other hand is more a zoom feature that selects a subset over all the dimensions but for specific values of the dimension.

Program slicing

- Program slicing is **a technique for simplifying programs by focusing on selected aspects of semantics**. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest.
- Program slice is a decomposition technique that extracts statements relevant to a particular computation from a program
- Program slicing describes a mechanism which allows the automatic generation of a slice

What is program slicing?

- Slicing criterion <s , v>
  - Where s specifies a location (statement s) and v specifies a variable (v)

- All statements affecting or affected by the variables mentioned in the slicing criterion becomes a part of the slice

What is program slicing?

- Program slice must satisfy the following conditions
  - Slice *S(V,n)* must be derived from *P* by deleting statements from *P*
  - Slice *S(V,n)* must be syntactically correct
  - For all executions of *P*, the value of *V* in the execution of *S(V,n)* just before the location *n* must be the same value of *V* in the execution of the program *P* just before location *n*

Program Slicing Tools

1. Program Slicing is not a viable manual approach
2. The more elaborate tools feature inter procedural slicing , useful for larger systems.
3. Program slicing is used to improve program comprehension required by maintenance programmers.

Program slicing tools

- CodeSurfer
  - Commercial product by GammaTech Inc.
  - GUI Based
  - Scripting language-Tk
- Unravel
  - Static program slicer developed at NIST
  - Slices ANSI C programs
  - Limitations are in the treatment of Unions, Forks and pointers to functions

2.3 Software Verification and Validation

- Introduction, Verification, Verification Workbench, Methods of Verification, Types of reviews on the basis od Stage Phase, Entities involved in verification, Reviews in testing lifecycle, Coverage in Verification, Concerns of Verification, Validation, Validation Workbench, Levels of Validation, Coverage in Validation, Acceptance Testing, Management of Verification and Validation, Software development verification and validation activities.

Quality planning at organization level

- Definition of processes , procedures , standards, guidelines etc for developing and testing
- Performing quality operations such as verification and validation
- Auditing work products
- Collecting metrics
- Defining actions for lacunae found

Verification

- Verification is a disciplined approach to evaluate whether a software product fulfills the requirements or conditions imposed on them by the standards or processes

- Prerequisites for verification

1. Training required by people for conducting verification
2. Standards , guidelines , tools to be used during verification
3. Verification Do and Check process definition

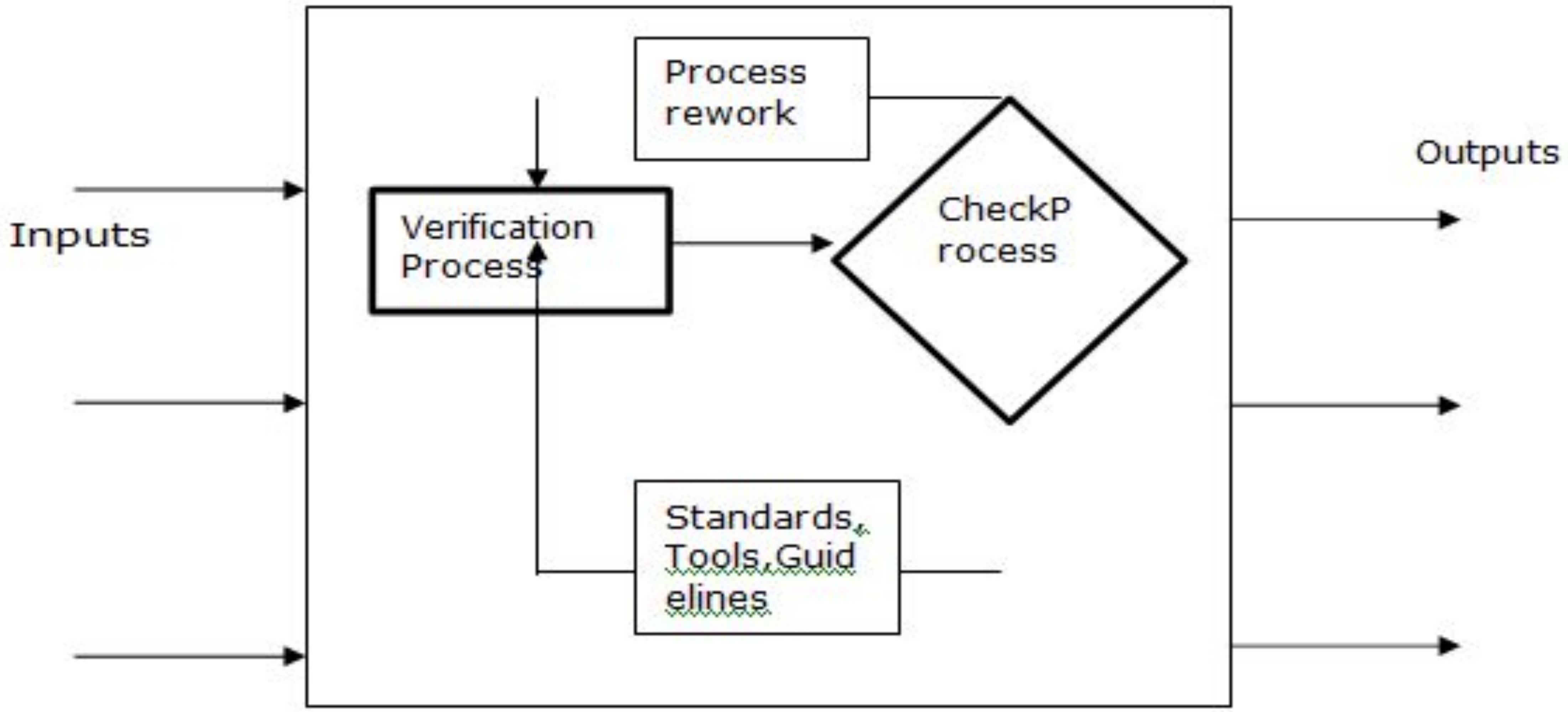
- Advantages of verification

1. Verification can confirm that the work product has followed the processes correctly
2. It can find defect in terms of deviations from standards easily
3. Location of the defect can be found
4. It can reduce the cost of finding and fixing the defects
5. It can locate the defect easily as work product under review is yet to be integrated
6. It can be used effectively for training people

- Disadvantage of verification

1. It cannot show that software developed is correct software
2. Actual working software may not be assessed by it

Verification workbench



Methods of verification

1. Self review
2. Peer reviews
3. Walkthrough
4. Inspection (Formal review)
5. Audits

Self review

- Advantages of self review
  - Self reviews are highly flexible
  - Self review is an excellent tool for self learning
  - There is no ego involved

- Disadvantages of self review
  - Approach or understanding related defects may not be found
  - People involved in self review may not conduct review in reality
  - Something which is implemented correctly in initial development may get changed due to personal issues

Peer reviews

1. Online peer review (Peer to peer review)
2. Offline peer review (Peer review)

- Advantages of peer review
  - Peer reviews are highly informal and unplanned in nature
  - There are no(less) egos or pride attached
  - Mostly the defects are discussed and decision is reached very informally
  - Peer review is excellent tool for educating peer about the artifact

- Disadvantages of peer review
  - Another person doing a review may or may not be expert in the artifacts under review
  - Peer may fix the defect without recording them
  - Sometimes the correct implementation may get replaced by the wrong one
  - Approach related defects may not be fixed

Superior review

- Advantages of superior review
  - Approach related defects can be found easily
  - Better way of doing the things learned by experience can be shared
- Disadvantages of superior review
  - Superior availability becomes bottleneck
  - It does not work efficiently when superior is also new to the domain / approach

Walkthroughs

- Advantages of walkthrough
  - Walkthrough is an excellent tool for collaboration
  - Walkthrough is also useful for training entire team at onetime
  - People understand what they can expect from the document / approach under consideration
  - Problems may get recorded and suggestions may be received from all team for improving the work product further
- Disadvantages of walkthrough
  - Availability of people can be an issue when the teams are large
  - Team may not be expert in giving the comments
  - Time can be a constraint where the schedules are very tight

Inspection

- Advantages of inspection
    - Expert's opinion is available as these people are present during review
    - Inspection must be very time effective as availability of experts is limited
    - Defects are recorded but solutions are not given
-
- Disadvantages of inspection
    - Experts may not be ready with review comments before inspection and time may not be utilized properly
    - Expert's opinion may vary from realities
    - Facilitator's job is critical

Guidelines for inspection

- Organization / project must plan for inspection well in advance
- Allocation of trained resources for facilitating inspection process, recording the opinions of experts, presenting artifacts etc is essential to maintain the agenda and inspection flow
- Expertise in terms of logistics arrangement is also needed
- Moderator/ leader must circulate the work product in advance
- Product must be clear, compiled, checked for grammar, formatting etc
- Inspection team maybe of 3 –5people
- Maximum time allocated must not exceed1–1.5hours
- It is moderator's responsibility to maintain the agenda of inspection
- Manager or superior of the author may not be involved in inspection
- Many organizations do not allow author to be present in inspection
- One must concentrate on locating defect and not on fixing them
- Classify and record defects in different categories
- Review inspection process for continuous improvement

Phases of inspection

- Planning for inspection
- Kick off of inspection
- Individual preparation
- Inspection meeting
- Decision on comments

- Roles and responsibilities
  - Manager
  - Moderator
  - Author
  - Reviewers (Checkers / Inspectors)
  - Scribe(or recorder)

Success factors for inspections

1. Each inspection must have a clear predefined objective
2. The right people must be involved in inspection
3. Defects found are welcomed and expressed objectively
4. People issues and psychological aspects are not dealt within an inspection process
5. Inspection techniques are applied that are suitable to the type and level of software work products and inspectors
6. Checklists or role playing are used if appropriate
7. Training is to be given in inspection techniques
8. Management support is essential for a good inspection process
9. There must be an emphasis on learning lessons and process improvement

Audit

1. Kick off audit
2. Periodic Software Quality Assurance audit
3. Phase end audit
4. Pre delivery audit
5. Product audit

Types of review on the basis of stage / phase

- In process review
- Milestone review
  - Phase end review
  - Periodic review
  - Percent completion

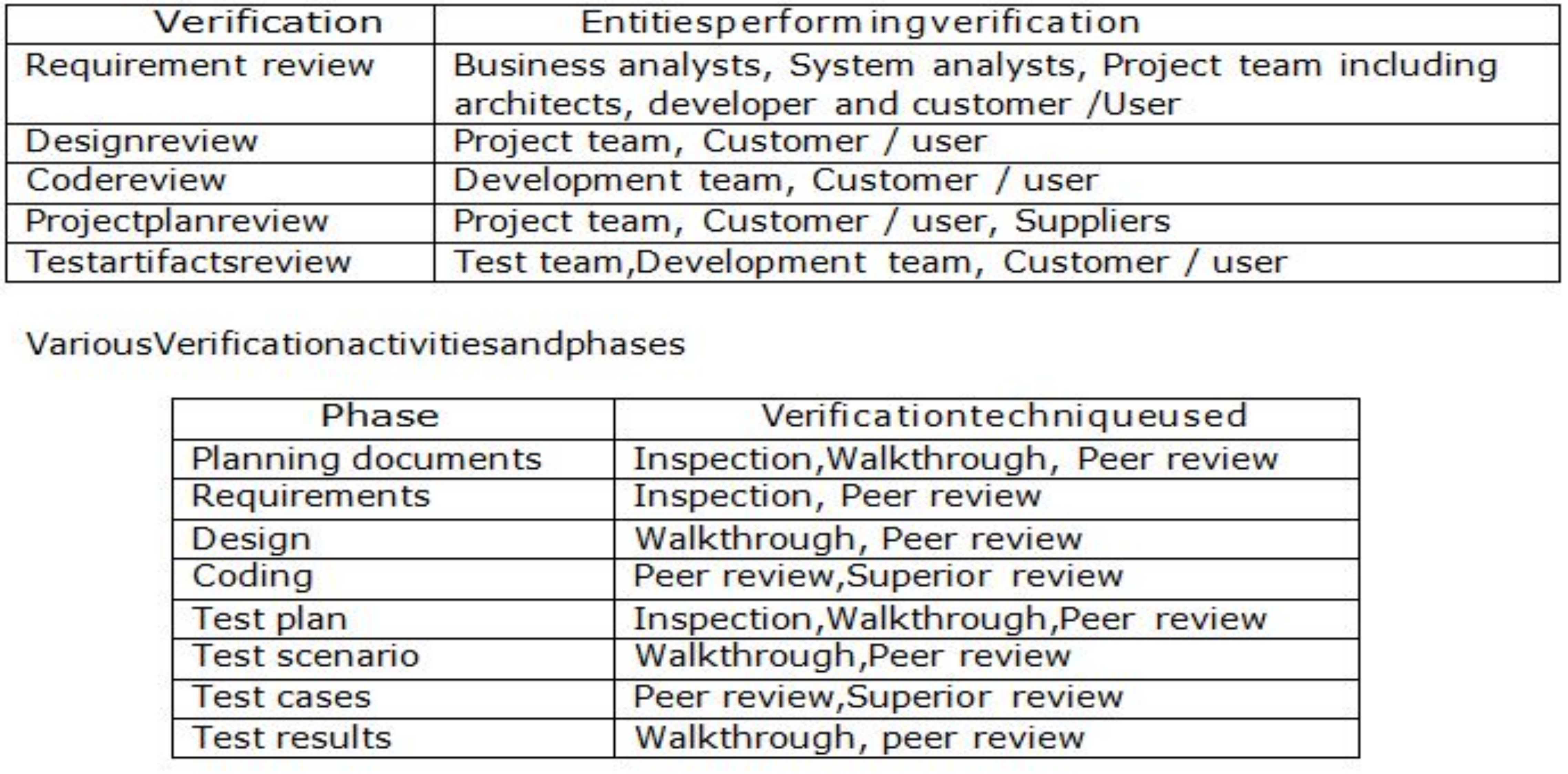
Process of In process review

- Work product, metrics etc undergoing review are created and submitted to the stakeholders well in advance
- Inputs are received from stakeholders to initiate various actions
- Risks identified by the project are reviewed and actions may be initiated as required
- Risks which no longer exist may be closed and risks which cannot affect the project are retired
- Any issues related to any stakeholder is noted and followup actions are initiated
- Review reports are generated at the end of review which acts as a foundation for next review

Post implementation review

- At the end of the project post implementation reviews are planned
- All the stakeholders of the project gather at a predefined time
- All the activities of project are reviewed and different stakeholders are expected to retrospect their part of work
- All reusable components are added in organization repository
- All risks identified during project execution are closed
- All the tasks defined in various plans are closed
- All team members are given a feedback about their performance in a project
- Skill sets of all team members are updated in organization database.
- Each and every activity is listed to identify if something went exceptionally good or exceptionally bad
- All project related metrics are reviewed.

Entities involved in verification



Verification	Entitiesperformingverification
Requirement review	Business analysts, System analysts, Project team including architects, developer and customer /User
Designreview	Project team, Customer / user
Codereview	Development team, Customer / user
Projectplanreview	Project team, Customer / user, Suppliers
Testartifactsreview	Test team,Development team, Customer / user

VariousVerificationactivitiesandphases

Phase	Verificationtechniqueused
Planning documents	Inspection,Walkthrough, Peer review
Requirements	Inspection, Peer review
Design	Walkthrough, Peer review
Coding	Peer review,Superior review
Test plan	Inspection,Walkthrough,Peer review
Test scenario	Walkthrough,Peer review
Test cases	Peer review,Superior review
Test results	Walkthrough, peer review

Reviews in testing lifecycle
Test readiness review

- Decision about the comments (Accept / Reject / Defer / Change) must be finalized and actions must be taken according to decisions
- All unit testing defects are closed
- Installation testing is successful
- Smoke testing is done to check whether the application is alive or not
- Sanity testing ,also called as build verification testing (BVT) is done to check whether the major functionalities of application are available to user or not
- Review of all the background arrangements including help files, release notes ,supporting documentation etc

Test completion review Coverage in verification

- Following coverage can be offered during verification
- Statement coverage
- Path coverage
- Decision coverage
- Decision–to–decision path coverage

Concerns of verification

- Use of right verification technique
- Integration of verification in SDLC
- Resources and skills available for verification

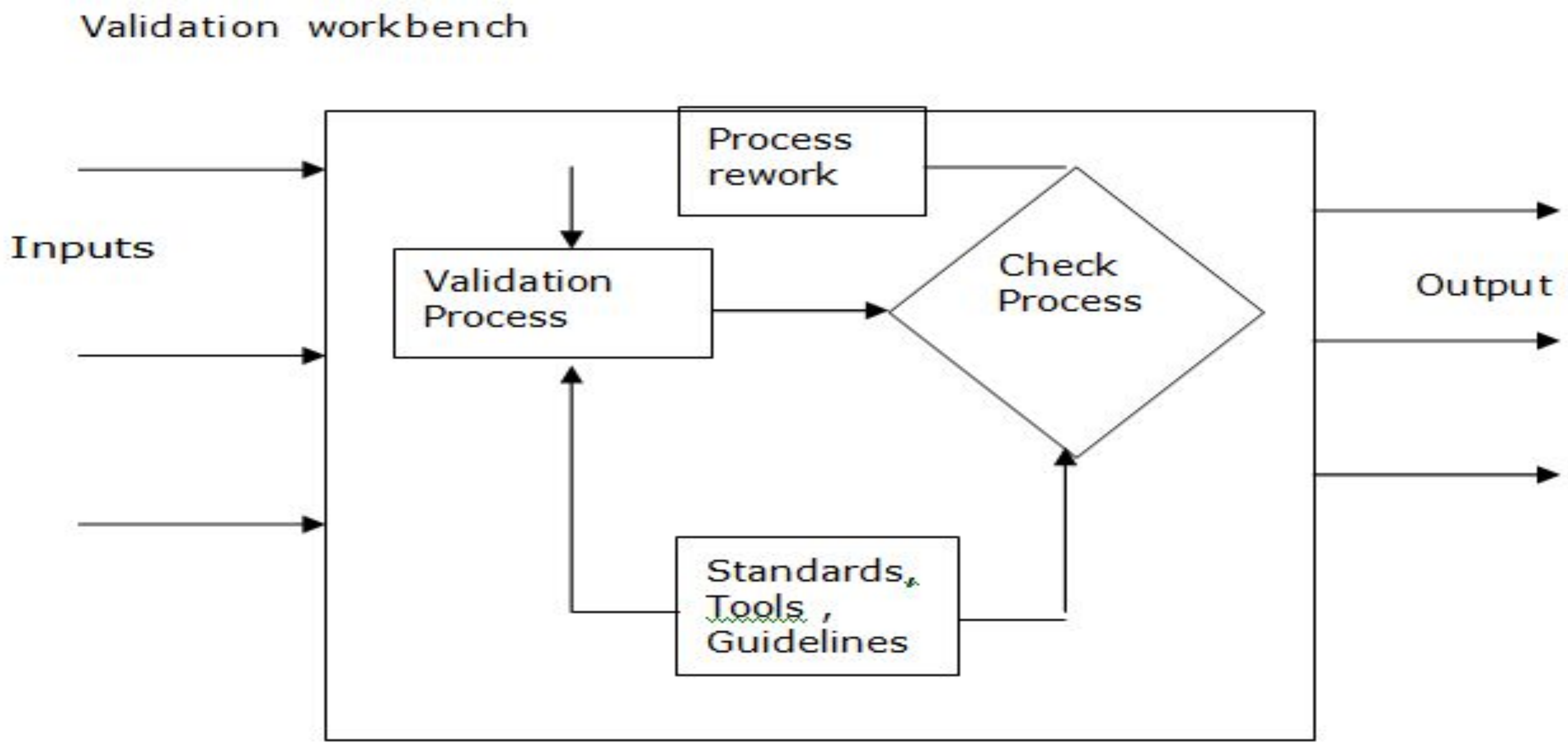
Validation

- Advantages of validation
  - It represents actual user interaction with the system without any consideration of internal structures or how the system is built
  - Validation is the only way to show that software is actually functioning
- Disadvantages of validation
  - No amount of testing can prove that software is not having defects
  - For unit testing and module testing, it needs stubs and drivers to be designed and to be used
  - Sometimes it may result in to redundant testing as tester are not aware of internal structure

Pre requisites for validation

- Training required for conducting validation . Training may include domain knowledge, knowledge about testing and knowledge about various test tools
- Standards ,guidelines , tools to be used during validation

Validation Do and Check process definition



Levels of validation

- Unit testing
- Integration testing
- Interface testing
- System testing

Coverage in validation (prioritization /slice based testing)

- Following are some of the famous coverage offered during validation.
  - Requirement coverage
  - Functionality coverage
  - Feature coverage

Acceptance testing

- Alpha testing
- Beta testing
- Gamma testing

Management of Verification and Validation(V&V)

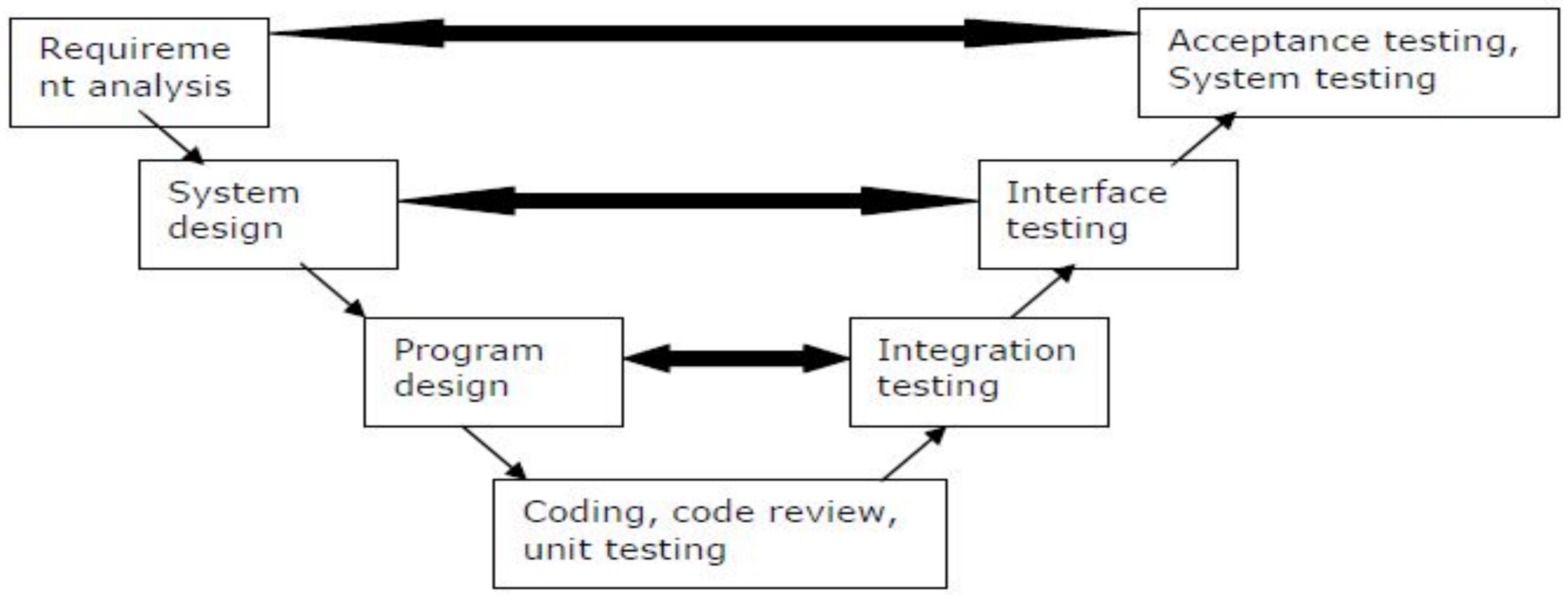
  - Defining the processes for Verification and Validation
- Software quality assurance process
- Software quality control process
- Software development process
- Software lifecycle definition
  - Prepare plans for execution of process
  - Initiate implementation plan
  - Monitor execution plan
  - Analyze problems discovered during execution
  - Report progress of the processes
  - Ensure product satisfies requirements

2.4 V-test Model

- Introduction,
- V-model for software,
- testing during Proposal stage,
- Testing during requirement stage,
- Testing during test planning phase,
- Testing during design phase,
- Testing during coding,
- VV Model,
- Critical Roles and Responsibilities.

V-TestModel

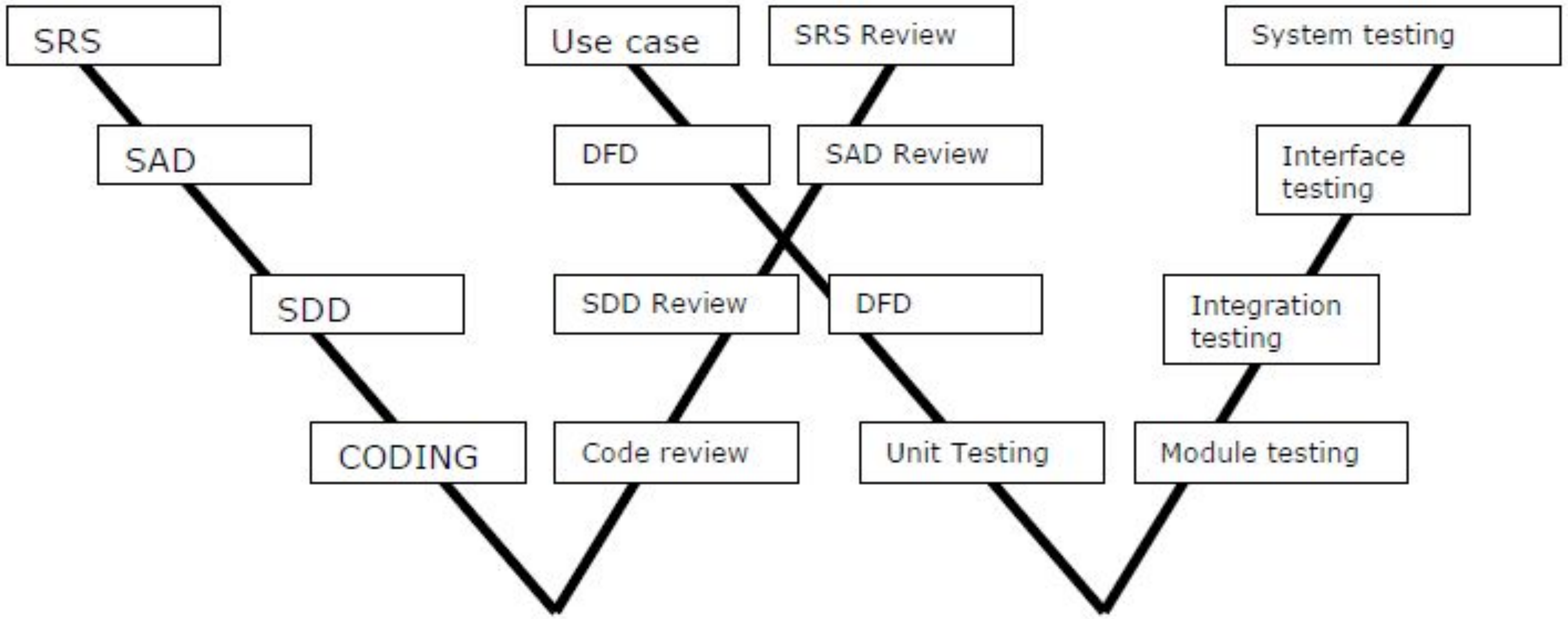
V model for software



Characteristics of good requirements

- Adequate
- Clear/Unambiguous
- Verifiable/Testable
- Measurable
- Feasible
- Not conflicting with eachother

VVmodel


SRS
SAD
SDD
CODING
Use case
DFD
SDD Review
Code review
SRS Review
SAD Review
DFD
Unit Testing
System testing
Interface testing
Integration testing
Module testing

Critical roles and responsibilities Development

- Project planning activities including requirement elicitation ,estimation ,project planning ,scheduling , definition of quality attributes required by the customer etc
- Resourcing may include identification and organization of adequate number of people , machines , hardware, software , tools etc
- Interacting with customer , other stakeholders as per project requirements
- Defining policies, procedures for creating development work, verification and validation activities to ensure that quality is built properly and delivering it to test team/ customer
- Supporting testing team by acknowledging the defects , giving inputs about requirements, giving executable on time, solving the queries raised by testing team or sending it to customer
- Doing development related activities such as capturing requirements, creating designs,coding,implementation etc

Testing

- Test planning including test strategy definition ,test planning ,test case writing etc
- Resourcing may include identification and organization of adequate number of people ,machines ,hardware ,software ,tools etc
- Interacting with customer ,other stakeholders as per project requirements
- Defining policies ,procedures for creating and executing tests as per test strategy and test plan
- Supporting development team by providing adequate information about the defects
- Doing acceptance testing related activities such as training ,mentoring to users from customer side before /during acceptance testing activities

Customer

- Specifying requirements and signing off requirement statement, designs etc
- Participating in acceptance testing as per roles and responsibilities defined in acceptance test plan
- Review and approve various artifacts as defined by contract or statement of work
- Participate in various reviews , walkthroughs, inspection activities