FAST

National University of Computer & Emerging Sciences

SOFTWARE TESTING

USAMA MUSHARAF

LECTURER (Department of Computer Science)

FAST-NUCES PESHAWAR

CONTENT

Software Testing

Testing Principles

Types of Testing

- Black Box
    - Black Box Testing Techniques
- White Box

Introduction

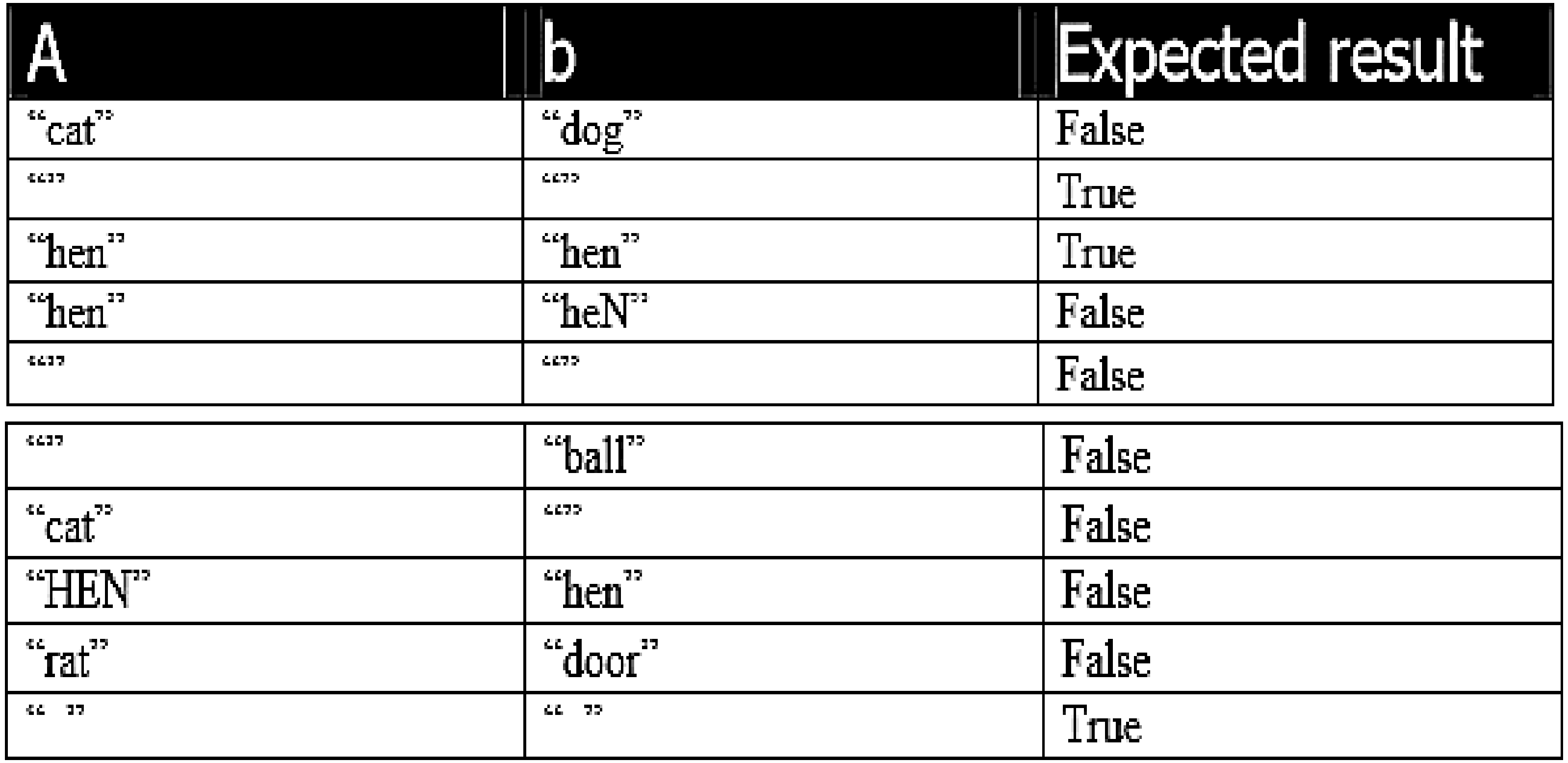
Testing has been described as the process of executing a program with the intention of finding errors.

Testing = process of searching for software errors

Software Testing (Definition)

Software testing is a formal process carried out by a **specialized testing team** in which a software unit, several integrated software units or an entire software package are examined by **running the programs on a computer.** All the associated tests are performed according to **approved test procedures** on **approved test cases**.

Example

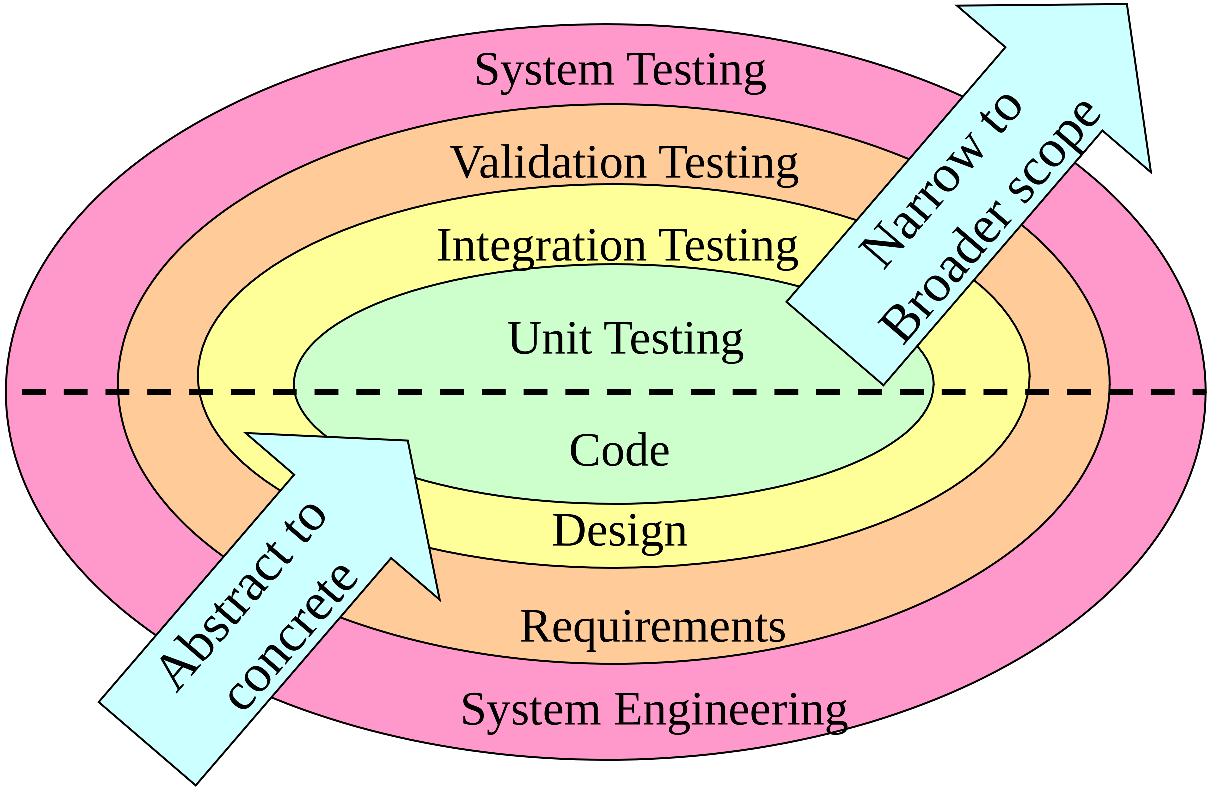


A	b	Expected result
“cat”	“dog”	False
“”	“”	True
“hen”	“hen”	True
“hen”	“heN”	False
“”	“”	False
“”	“ball”	False
“cat”	“”	False
“HEN”	“hen”	False
“rat”	“door”	False
“ ”	“ ”	True

Example

```
bool isStringsEqual(char a[], char b[]) {

    bool result;

    if (strlen(a) != strlen(b))
    {
      result = false;
    }
    else {
        for ( int i =0; i < strlen(a); i++)
        {
             if ( a[i] == b[i] )
            {  result = true;      }
            else
             { result = false;  }
        }

         }

   return result;
}
```

A Strategy for Testing Software



Testing Principles

1. All tests should be traceable to customer requirements
   - The objective of software testing is to find defects. It follows that the most severe defects are those that cause systems to fail to meet their requirements

2. Tests should be planned long before testing begins.
   - Test planning can begin as soon as the requirements model is complete

3. The Pareto (80-20) principle applies to software testing
   - 80% of all defects found will likely be traced to 20% of modules. These error-prone modules should be isolated and tested thoroughly.

Testing Principles

4. The testing should begin “in the small” and progress toward testing “in the large”.

- The first tests planned and executed focus on individual components. As testing progresses, focus shifts to integrated clusters of components.

5. Exhaustive testing is not possible.

- It is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Levels of Specifications

There are usually at least three levels of software specification documents in large systems:

1. **Functional specifications** (or requirements) give a precise description of the required behavior of the system – what the software should do, not how it should do it – may also describe constraints on how this can be achieved.

2. **Design specifications** describe the architecture of the design to implement the functional specification – the components of the software and how they are to relate to one another.

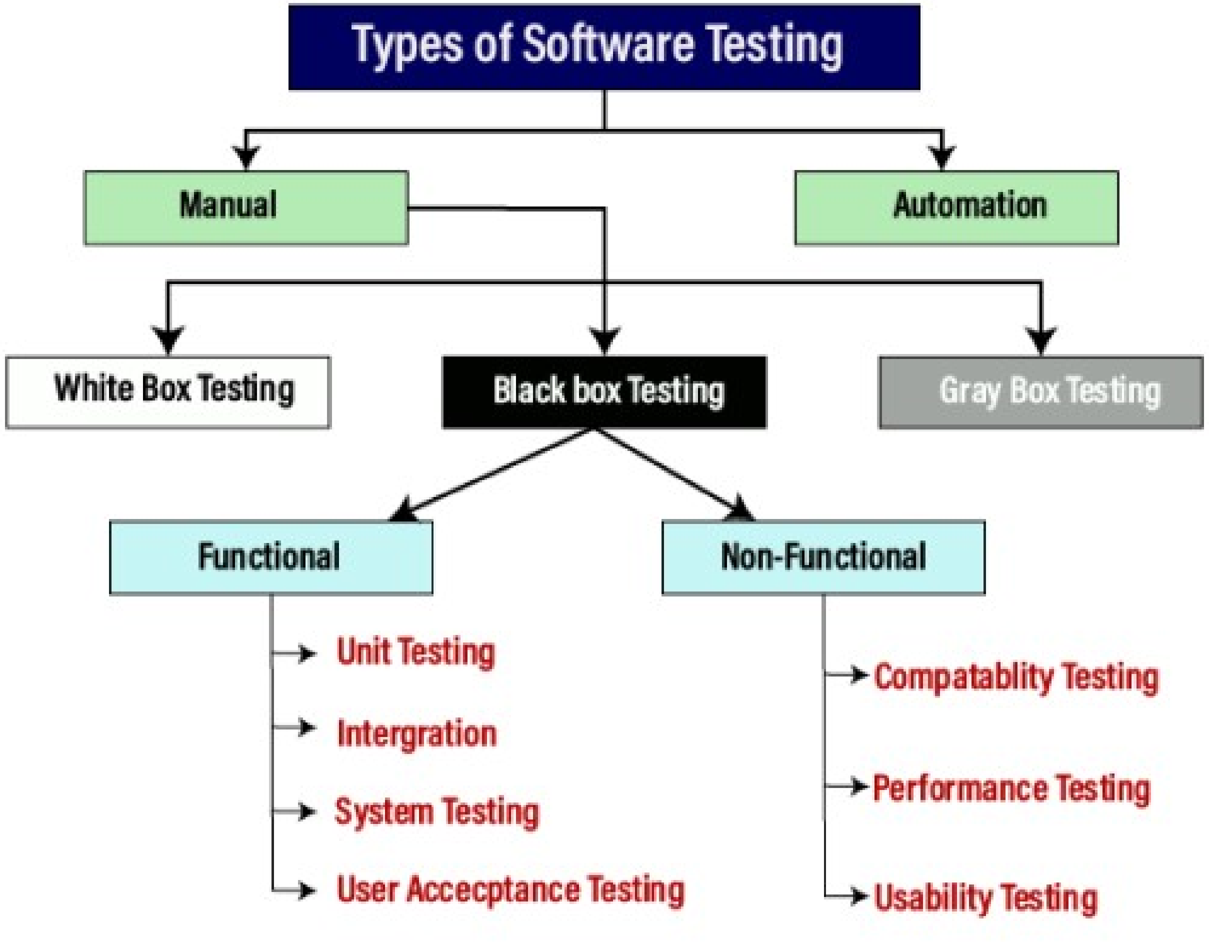
3. **Detailed design specifications** describe how each component of the architecture is to be implemented – down to the individual code units.

TYPES OF SOFTWARE TESTING

Manual vs Automation

In manual testing (as the name suggests), test cases are executed manually (by a human, that is) without any support from tools or scripts.

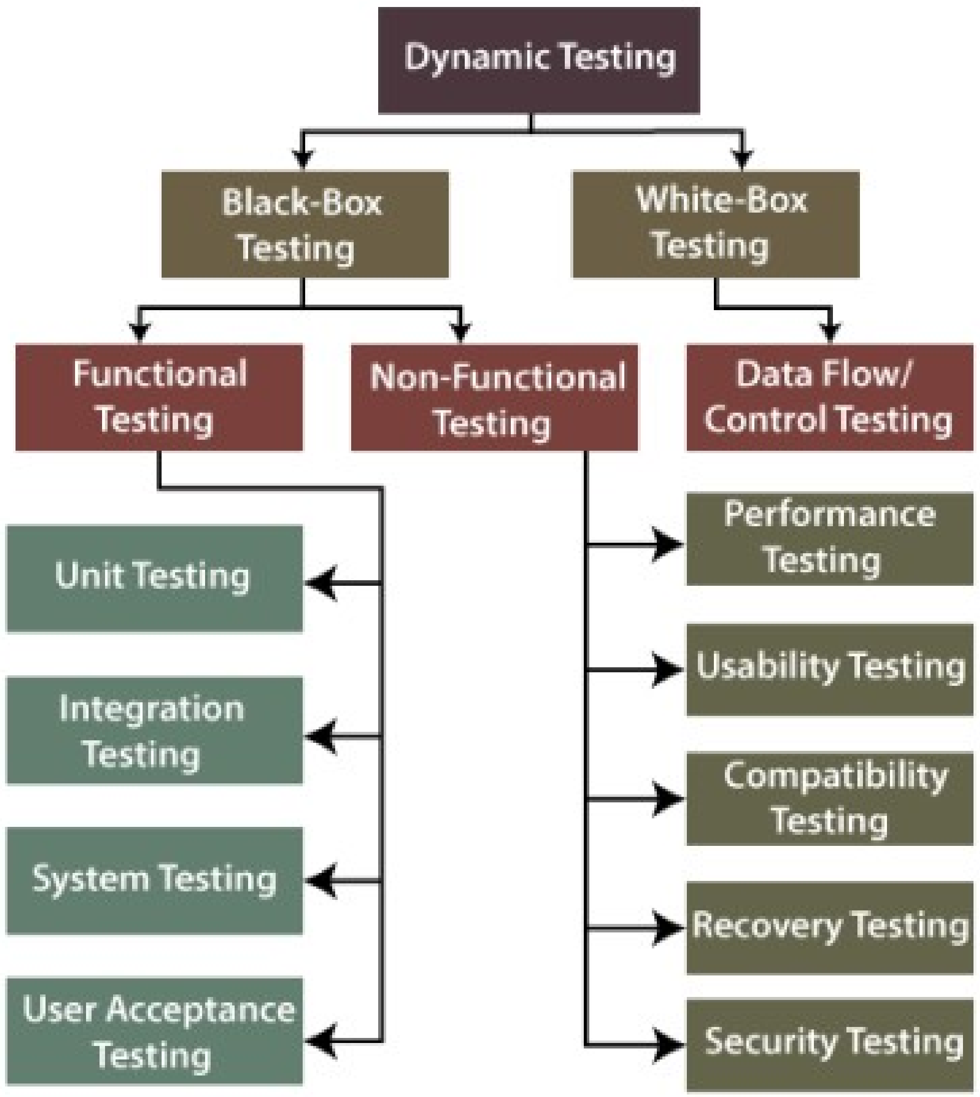
But with automated testing, test cases are executed with the assistance of tools, scripts, and software.



Static vs Dynamic

Static Testing is a type of software testing in which software application is tested without code execution. Manual reviews of code, requirement documents and document design are done in order to find the errors.

The main objective of static testing is to improve the quality of software applications by finding errors in early stages of software development process.



Static Testing

To avoid the errors, we will execute Static testing in the initial stage of development because it is easier to identify the sources of errors, and it can fix easily.

We can do some of the following important activities while performing static testing:

- Business requirement review (functional requirement, Use case reviews)
- Design review (architecture + design) (non-functional requirements)
- The test documentation review

Static Testing

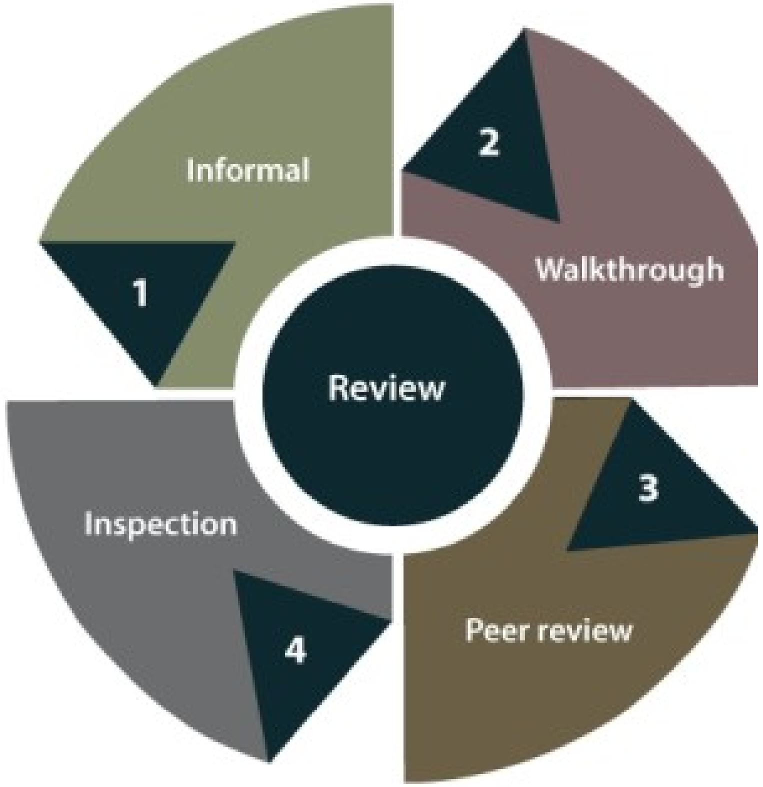
In static testing, reviews can be divided into four different parts, which are as follows:

Informal reviews

In informal review, the document designer place the contents in front of viewers, and everyone gives their view; therefore, bugs are acknowledged in the early stage.

Walkthrough

Generally, the walkthrough review is used to performed by a skilled person or expert to verify the bugs. Therefore, there might not be problem in the development or testing phase.



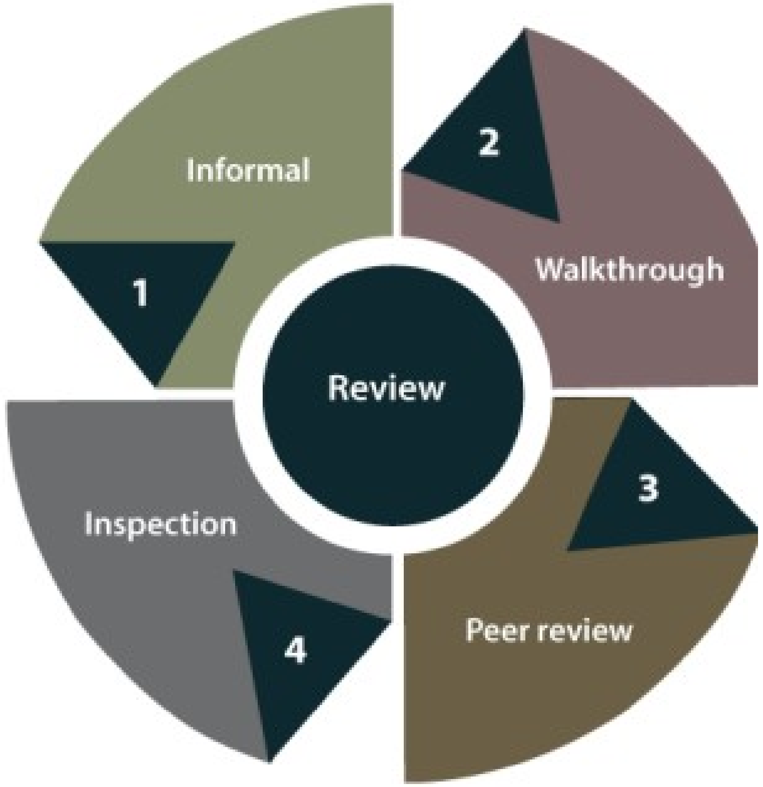
Static Testing

Peer review

In Peer review, we can check one another's documents to find and resolve the bugs, which is generally done in a team.

Inspection

In review, the inspection is essentially verifying the document by the higher authority, for example, the verification of SRS [software requirement specifications] document.



BLACK BOX TESTING

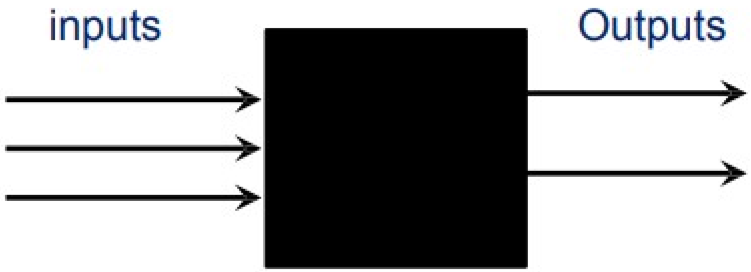
Black Box Testing

- A strategy in which testing is based on requirements and specifications.

- In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed.

- No knowledge of internal design or code required.

- Tests are data driven.



BLACK BOX TESTING TECHNIQUES

Black Box Testing Techniques

- Exhaustive testing
- Equivalence class testing (Equivalence Partitioning)
- Boundary value analysis
- Decision table testing
- State-Transition testing

1. Exhaustive testing

Definition: testing with every member of the input value space

Input value space: the set of all possible input values to the program

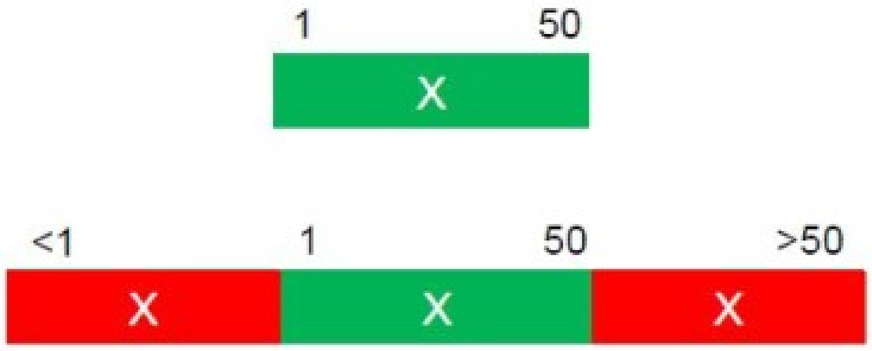
- Consider an application in which a password field that accepts 3 characters, with no consecutive repeating entries. Hence, there are 26 * 26 * 26 input permutations for alphabets only. Including special characters and standard characters, there are much more combinations. So, there are 256 * 256 * 256 input combinations.

2. Equivalence Class Testing/ Portioning

Equivalence Class Testing is when you have a number of test items (e.g. values) that you want to test but because of cost (time/money) you do not have time to test them all.

Therefore you group the test item into class where all items in each class are suppose to behave exactly the same.

The theory is that you only need to test one of each item to make sure the system works.

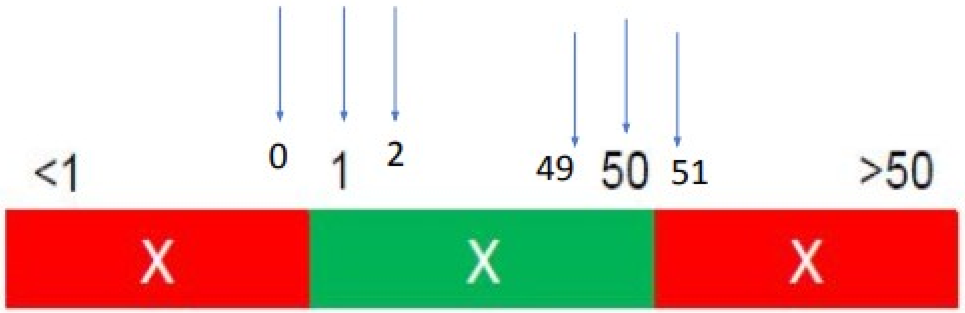


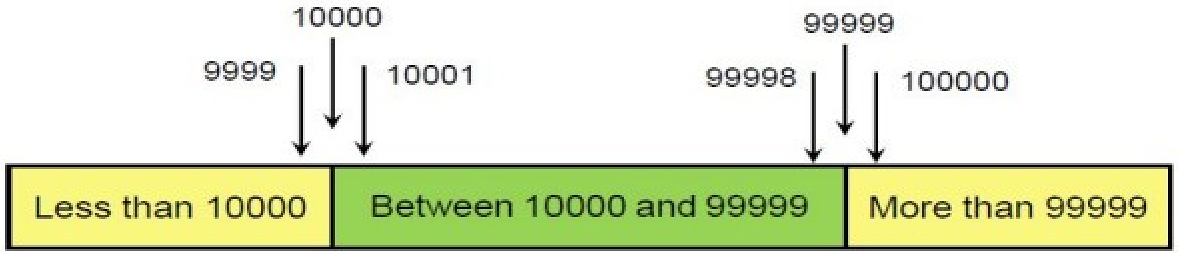
3. Boundary Value Analysis

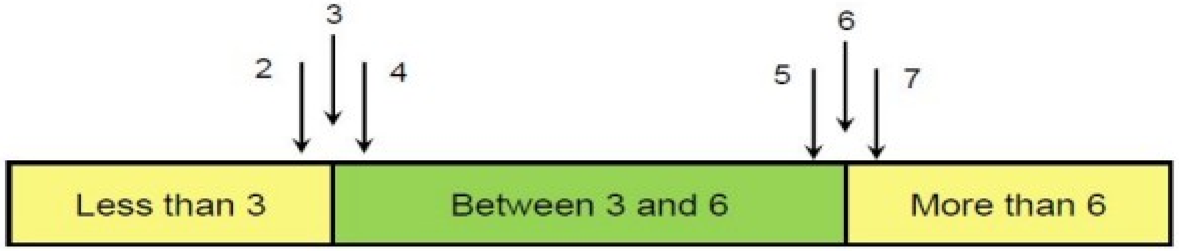
Maximum Defects occurs on Boundaries

1.Identify the **Equivalence Classes** and Identify the boundaries of each **Equivalence Class.**

2.Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.







Why to go with Boundary Value Analysis?

Consider an example where a developer writes code for an amount text field which will accept and transfer values only from 100 to 5000.

The test engineer checks it by entering 99 into the amount text field and then clicks on the transfer button. It will show an error message as 99 is an invalid test case, because the boundary values are already set as 100 and 5000.

Since 99 is less than 100, the text field will not transfer the amount.

4. Decision Table Testing

An tool for capturing certain kinds of system requirements and documenting internal system design.

Record complex business rules that a system must implement.

Also can serve as a guide to creating test cases.

4. Decision Table Testing

Condition

- Condition describe the conditions or factors that will affect the decision or policy.
- They are listed in the upper section of the decision table.

Action

- Action describe, in the form of statements, the possible policy actions or decisions
- They are listed in the lower section of the decision table.

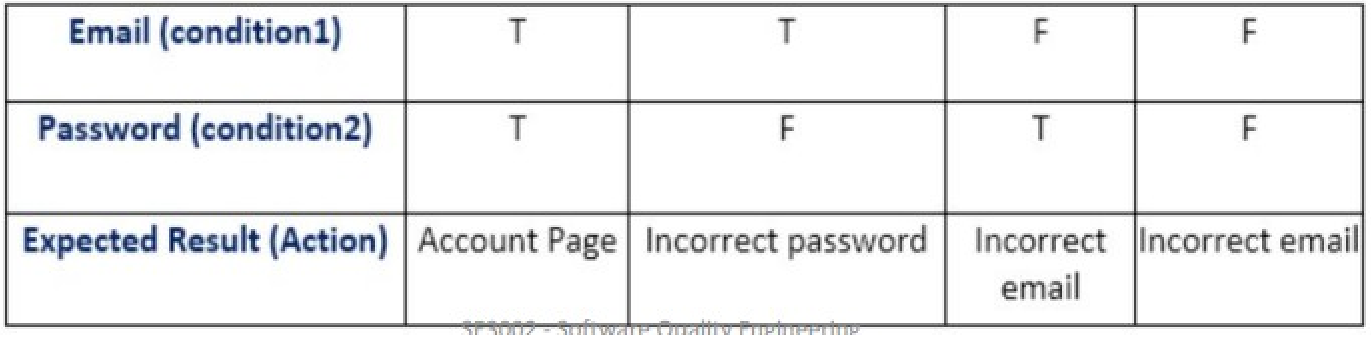
Rules

- Rules describe which actions are to be taken under a specific combination of conditions.

4. Decision Table Testing

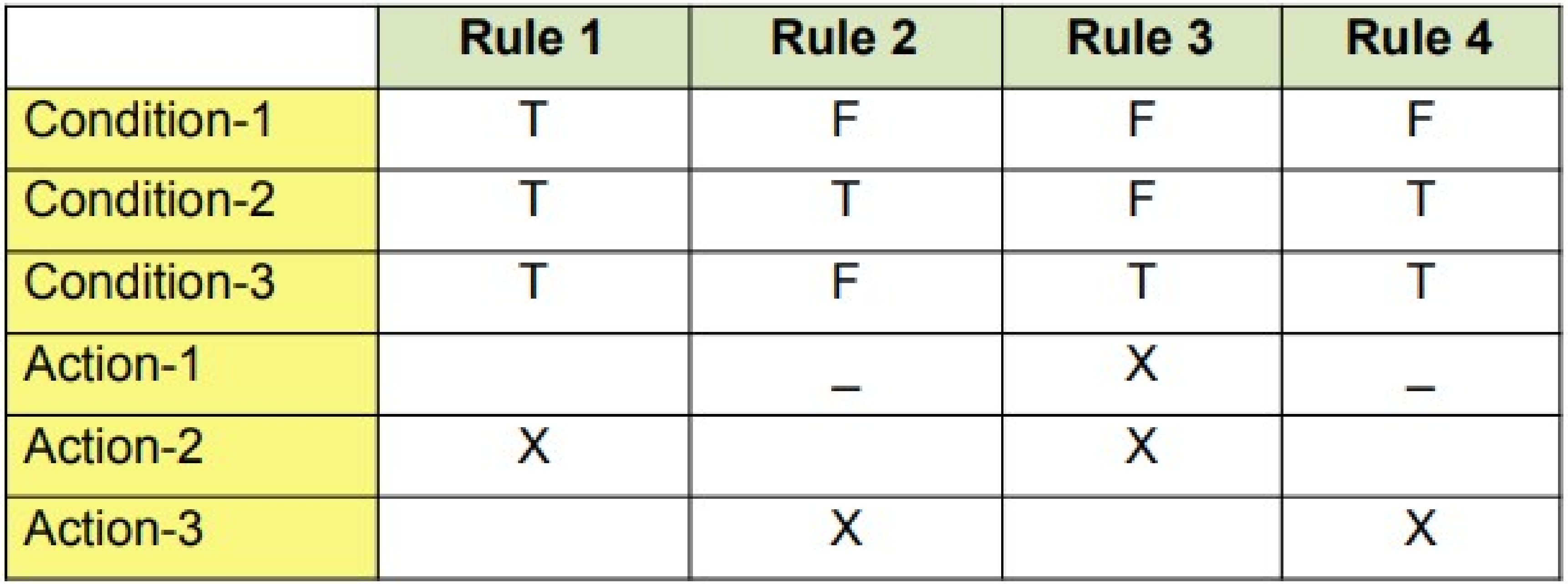
Most of us use an email account, and when you want to use an email account, for this you need to enter the email and its associated password.

- 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."



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

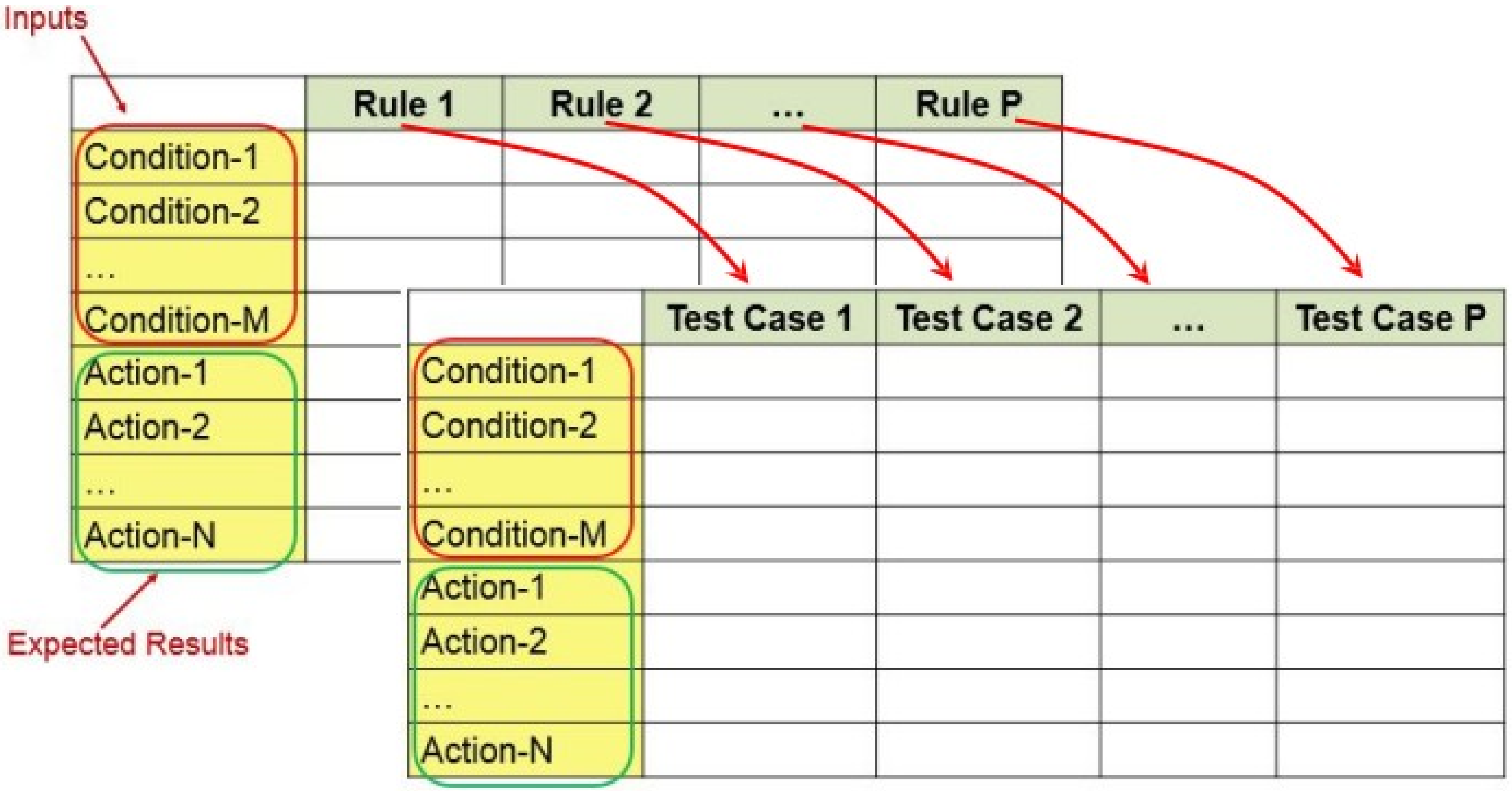
4. Decision Table



	Rule 1	Rule 2	Rule 3	Rule 4
Condition-1	T	F	F	F
Condition-2	T	T	F	T
Condition-3	T	F	T	T
Action-1		–	X	–
Action-2	X		X	
Action-3		X		X

Decision Table Vs Test Case Table

A decision table converted to a test case.



	Rule 1	Rule 2	…	Rule P
Condition-1				
Condition-2				
…				
Condition-M				
Action-1				
Action-2				
…				
Action-N				

	Test Case 1	Test Case 2	…	Test Case P
Condition-1				
Condition-2				
…				
Condition-M				
Action-1				
Action-2				
…				
Action-N				

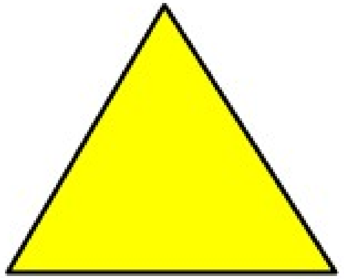
Decision Table – Triangle Program

- A program with three input integers, a, b, and c
- If a triangle then integers a, b, and c must satisfy:

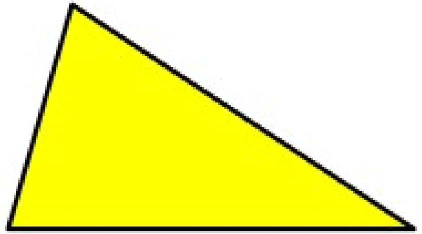
$$a < b + c$$

$$b < a + c$$

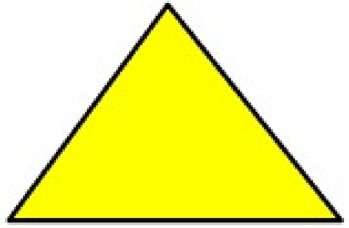
$$c < a + b$$



Isosceles

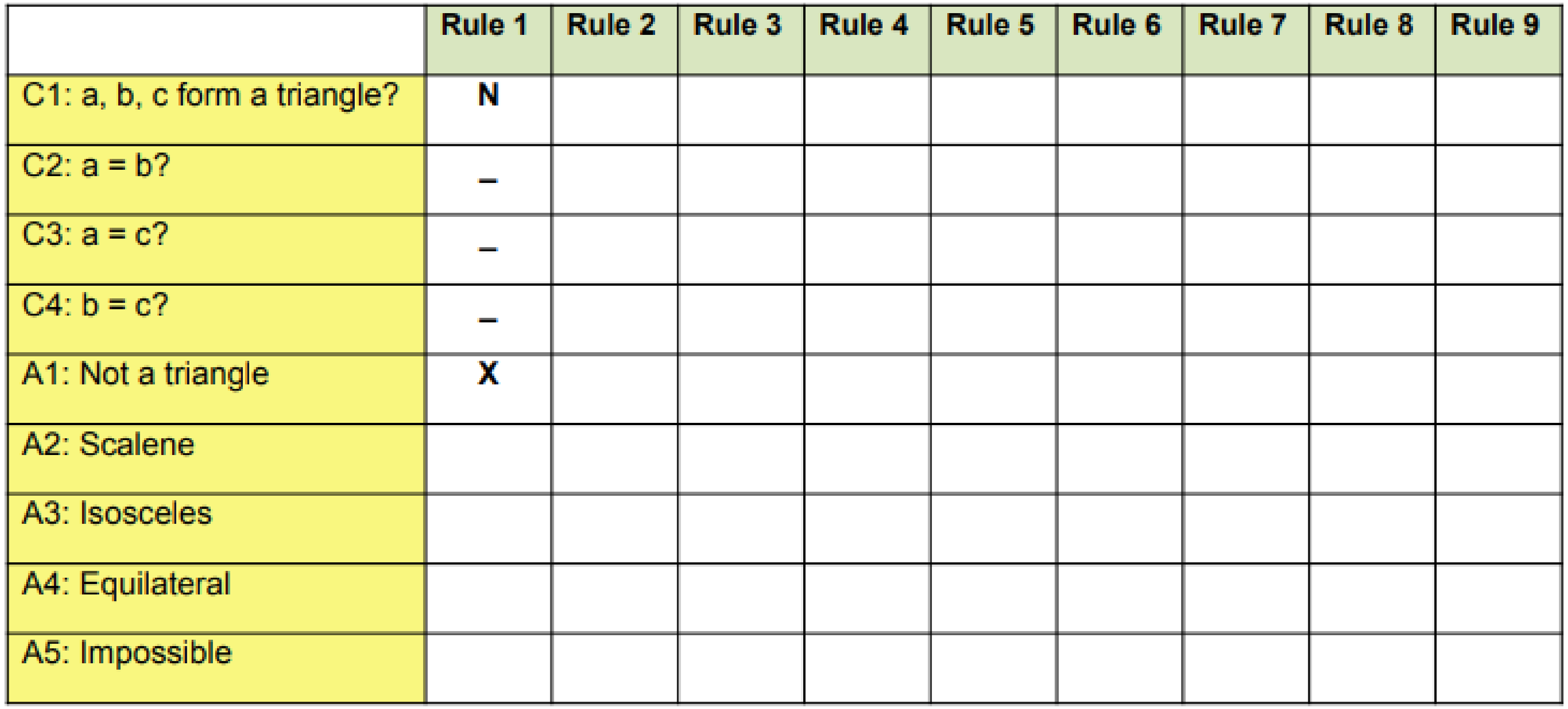


Scalene ($a \neq b \neq c$)



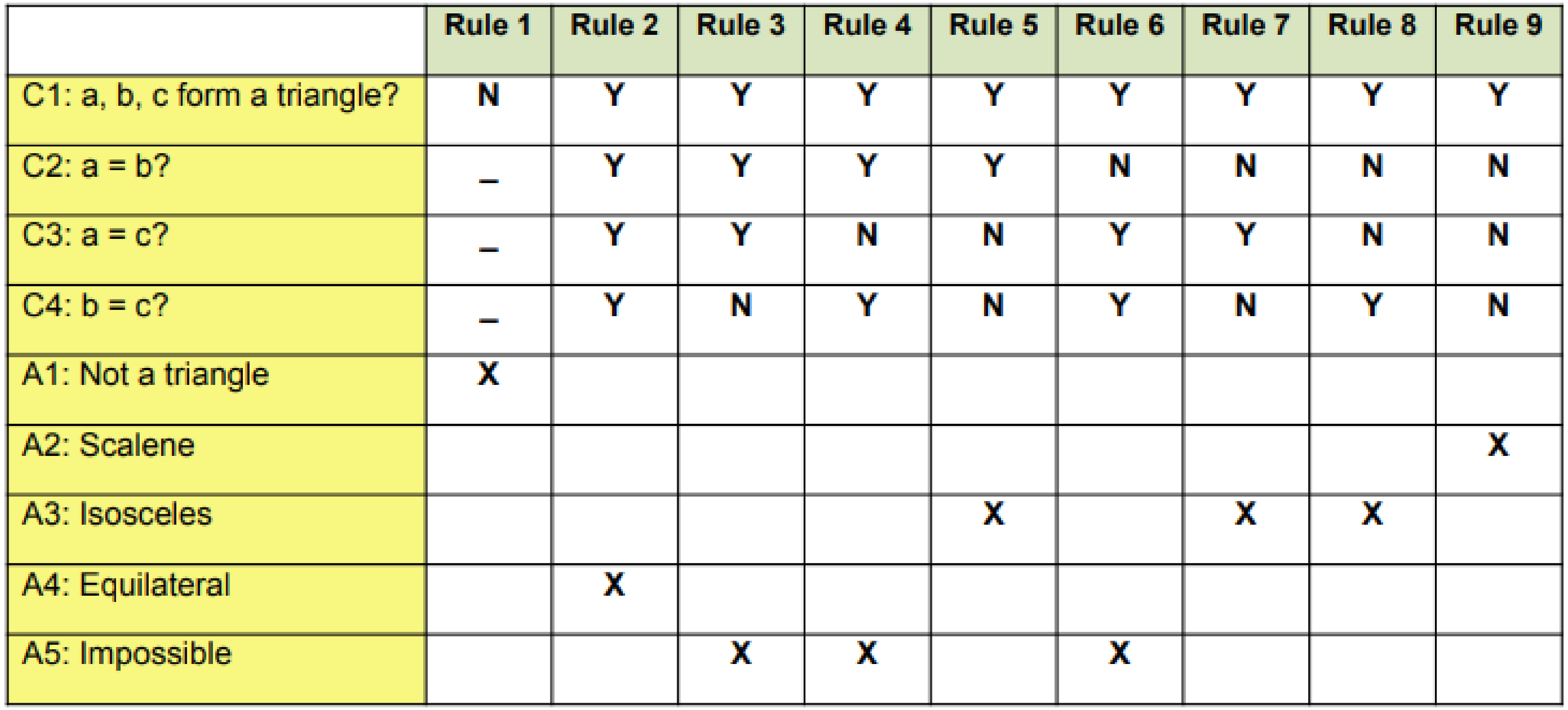
Equilateral ($a = b = c$)

Decision Table – Triangle Program



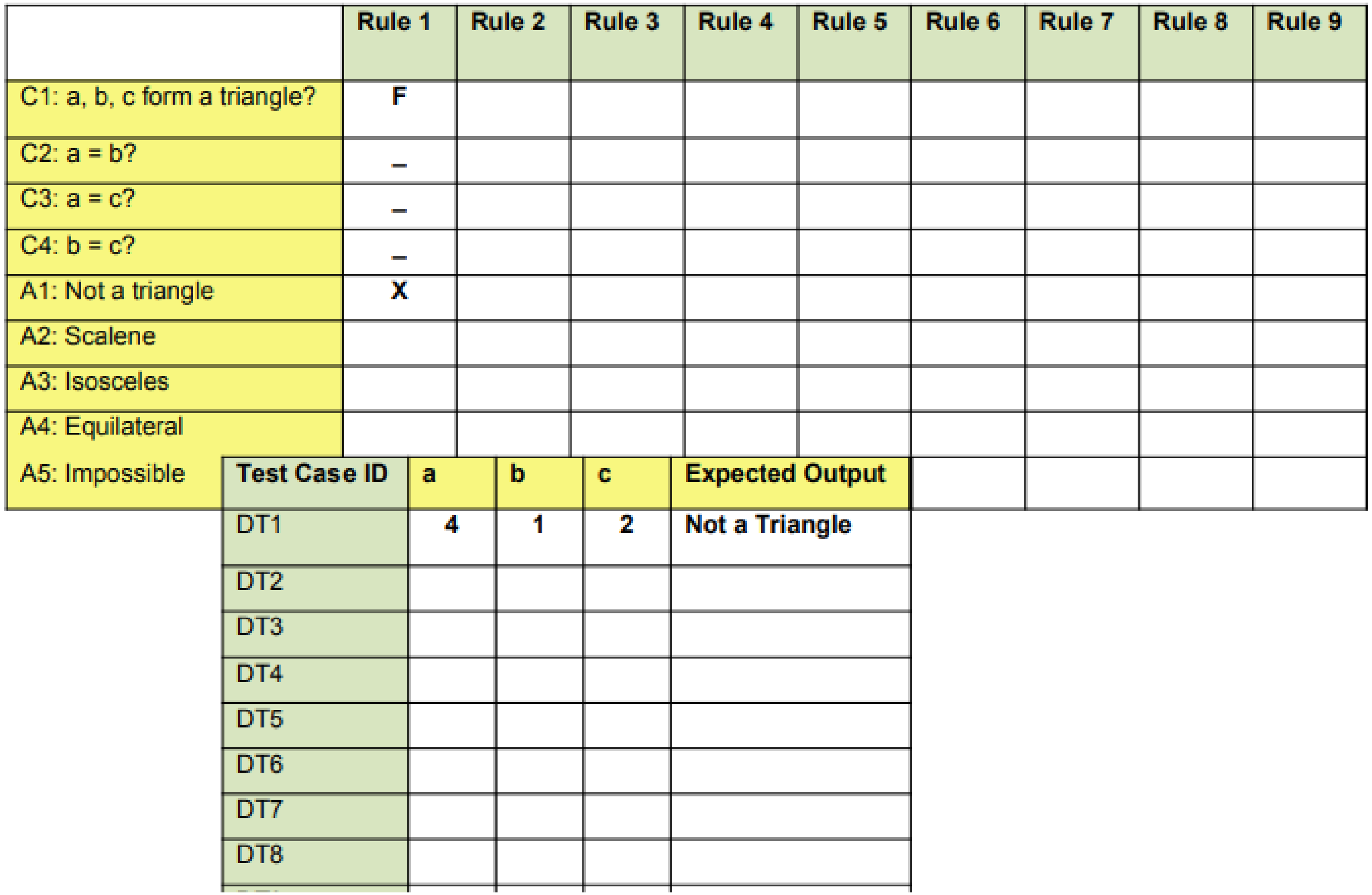
	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9
C1: a, b, c form a triangle?	**N**								
C2: a = b?	–								
C3: a = c?	–								
C4: b = c?	–								
A1: Not a triangle	**X**								
A2: Scalene									
A3: Isosceles									
A4: Equilateral									
A5: Impossible									

Decision Table – Triangle Program



	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9
C1: a, b, c form a triangle?	N	Y	Y	Y	Y	Y	Y	Y	Y
C2: a = b?	_	Y	Y	Y	Y	N	N	N	N
C3: a = c?	_	Y	Y	N	N	Y	Y	N	N
C4: b = c?	_	Y	N	Y	N	Y	N	Y	N
A1: Not a triangle	X								
A2: Scalene									X
A3: Isosceles					X		X	X	
A4: Equilateral		X							
A5: Impossible			X	X		X			

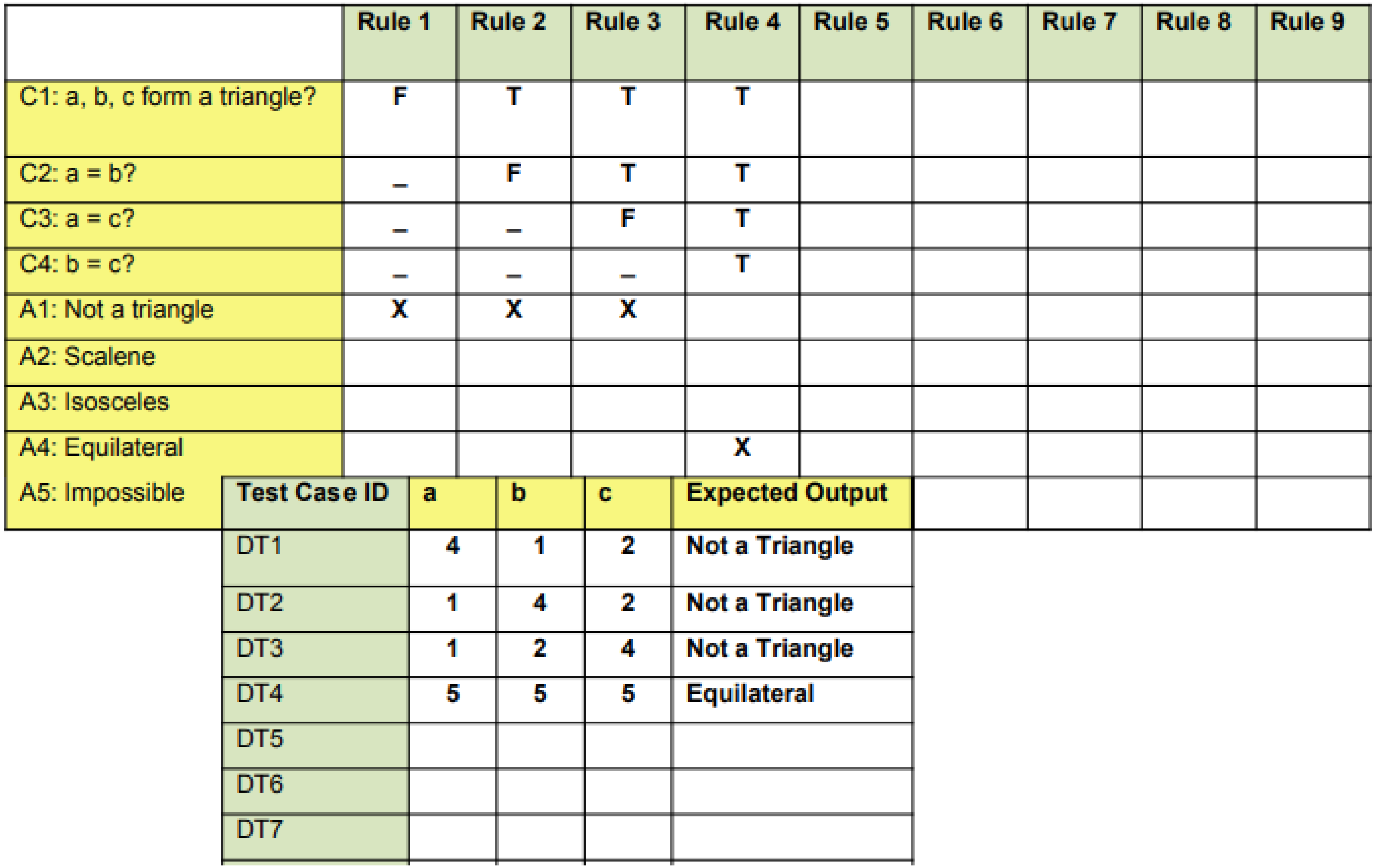
Decision Table vs Test Case Table



	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9
C1: a, b, c form a triangle?	F								
C2: a = b?	–								
C3: a = c?	–								
C4: b = c?	–								
A1: Not a triangle	X								
A2: Scalene									
A3: Isosceles									
A4: Equilateral									
A5: Impossible									

Test Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2				
DT3				
DT4				
DT5				
DT6				
DT7				
DT8				

Decision Table vs Test Case Table



	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8	Rule 9
C1: a, b, c form a triangle?	F	T	T	T					
C2: a = b?	_	F	T	T					
C3: a = c?	_	_	F	T					
C4: b = c?	_	_	_	T					
A1: Not a triangle	X	X	X						
A2: Scalene									
A3: Isosceles									
A4: Equilateral				X					
A5: Impossible									

Test 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				
DT6				
DT7				

Applicability

Decision table testing can be used whenever the system must implement complex business rules when these rules can be represented as a combination of conditions and when these conditions have discrete actions associated with them.

5. State Transition Testing

State Transition Testing is a black box testing technique in which changes made in input conditions cause state changes or output changes in the Application under Test.

State transition testing helps to analyze behavior of an application for different input conditions.

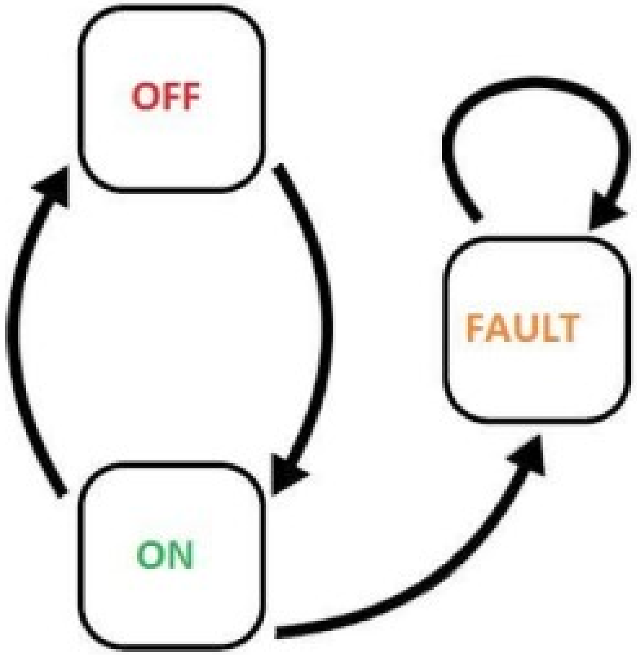
Testers can provide positive and negative input test values and record the system behavior.

5. State Transition Testing

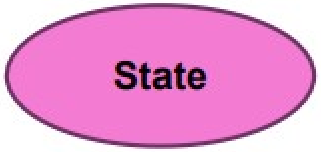
Models each state a system can exist in

- Models each state transition

- States
  - – Start State
  - – Input
  - – Output
  - – Finish State



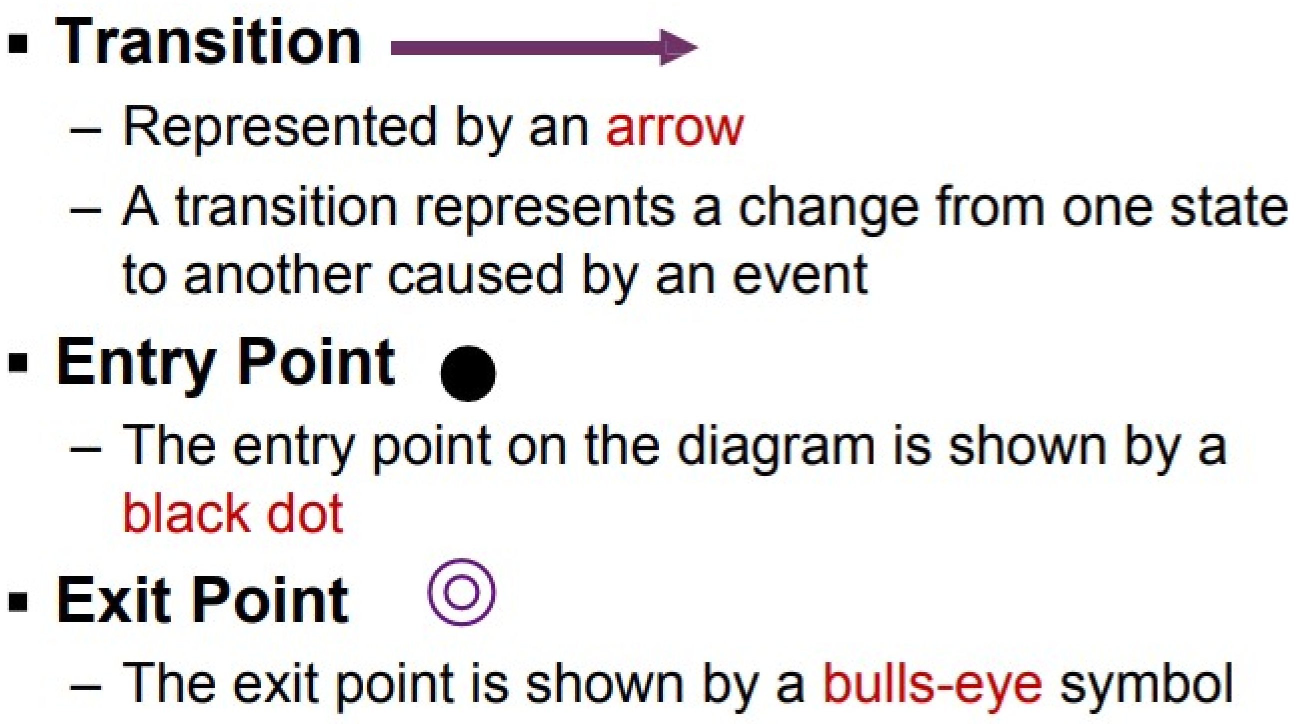
5. State Transition Testing



State –
Represented by a circle or oval shape.

- A state is a condition in which a system is waiting for one or more events.
- States "remember" inputs the system has received in the past and define how the system should respond to subsequent events when they occur.
- These events may cause state-transitions and/or initiate actions.

5. State Transition Testing



- **Transition** →
  - Represented by an arrow
  - A transition represents a change from one state to another caused by an event
- **Entry Point** ●
  - The entry point on the diagram is shown by a black dot
- **Exit Point** ◎
  - The exit point is shown by a bulls-eye symbol

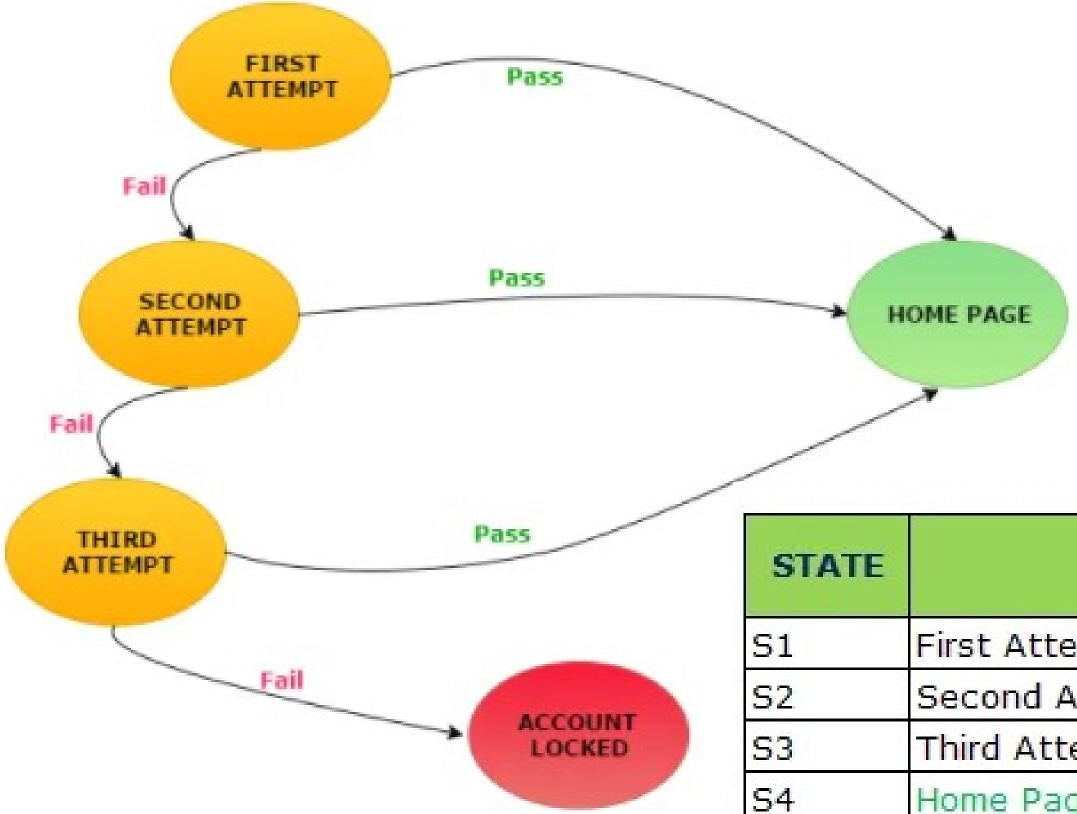
5. State Transition Testing

Action

An action is an operation initiated because of a state change.

Often these actions cause something to be created that are outputs of the system.

Actions occur on transitions between states



STATE	LOGIN	CORRENT PASSWORD	INCORRECT PASSWORD
S1	First Attempt	S4	S2
S2	Second Attempt	S4	S3
S3	Third Attempt	S4	S5
S4	Home Page		
S5	Display a message as "Account Locked, please consult Administrator"		

TYPES OF BLACK BOX TESTING

Types of black box testing

1. Functional testing
2. System testing
3. End-to-end testing
4. Sanity testing
5. Regression testing
6. Acceptance testing
7. Load testing
8. Stress testing
9. Install/uninstall testing
10. Recovery testing
11. Compatibility testing
12. Comparison testing
13. Alpha testing
14. Beta testing
15. Mutation testing

Functional testing

- Black box type testing geared to functional requirements of an application.

System testing

- Black box type testing that is based on overall requirements specifications; covering all combined parts of the system.

End-to-end testing

- Similar to system testing; involves testing of a complete application environment in a situation that mimics real-world use.

Sanity testing

- Initial effort to determine if a new software version is performing well enough to accept it for a major testing effort.

Regression testing

- Re-testing after fixes or modifications of the software or its environment.

Acceptance testing

- Final testing based on specifications of the end-user or customer.

Load testing

- Testing an application under heavy loads.
- Eg. Testing of a web site under a range of loads to determine, when the system response time degraded or fails.

Stress Testing

- Testing under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database etc.
- Term often used interchangeably with ‘load’ and ‘performance’ testing.

Performance testing

- Testing how well an application complies to performance requirements

Install/uninstall testing

- Testing of full, partial or upgrade install/uninstall process.

Recovery testing

- Testing how well a system recovers from crashes, HW failures or other problems.

Compatibility testing

- Testing how well software performs in a particular HW/SW/OS/NW environment.

Comparison testing

- Comparing SW strengths and weakness to competing products.

Alpha testing

- Testing done when development is nearing completion; minor design changes may still be made as a result of such testing.

Beta-testing

- Testing when development and testing are essentially completed and final bugs and problems need to be found before release.

Mutation testing

- To determining if a set of test data or test cases is useful, by deliberately introducing various bugs.
- Re-testing with the original test data/cases to determine if the bugs are detected.

WHITE BOX TESTING

White box testing / Structural testing

Based on knowledge of internal logic of an application's code

Based on coverage of code statements, branches, paths, conditions.

Tests are logic driven.

Control Flow Testing

Based on the flow of control in the program.

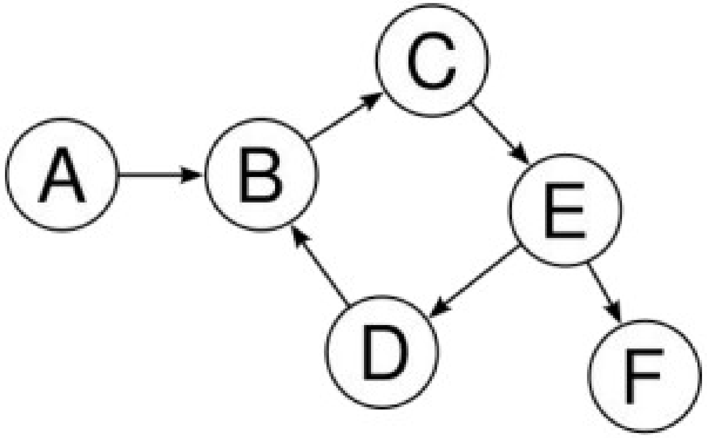
- Logical decisions
- Loops
- Execution paths

Coverage metrics

- Measure of how complete the test cases are.

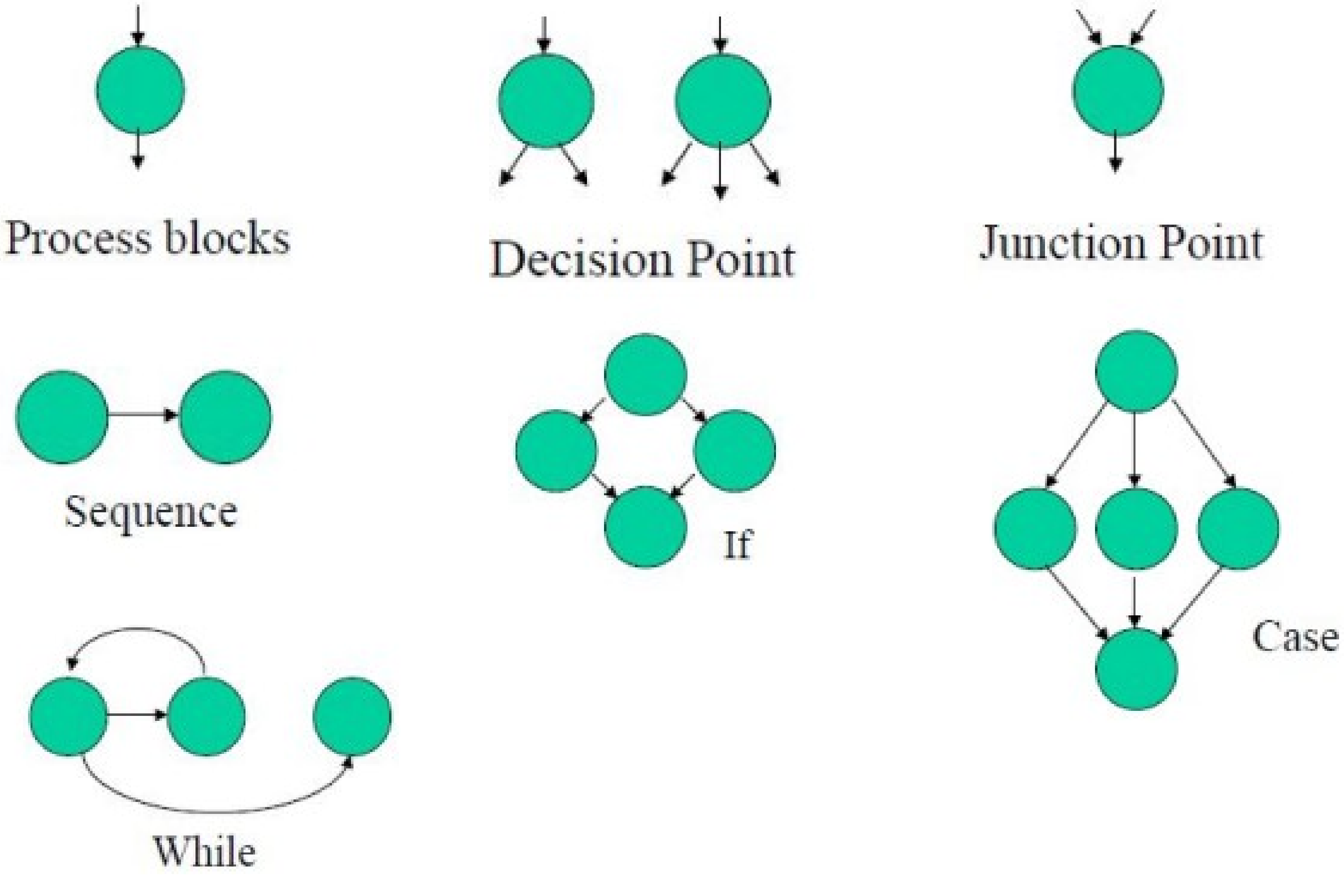
Control Flow Graph

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.



Directed Graph

Control Flow Graph



Steps to Draw Control Flow Graph

1. Number all the statements of a program.

2. Numbered statements: represent nodes of the control flow graph.

3. An edge from one node to another node exists: if execution of the statement representing the first node can result in transfer of control to the other node.

Program flow graph
Basic Control Flow Graph

```
class IfStatement {

public static void main(String[] args) {
    int number = 10;
        if (number > 0) {
             System.out.println("The number is positive.");
                 }
        else {
             System.out.println("The number is negative.");
                       }
          System.out.println("Statement outside if block");
     }
}
```

Coverage:

Statement Coverage:

In this scheme, statements of the code are tested for a successful test that checks all the statements lying on the path of a successful scenario.

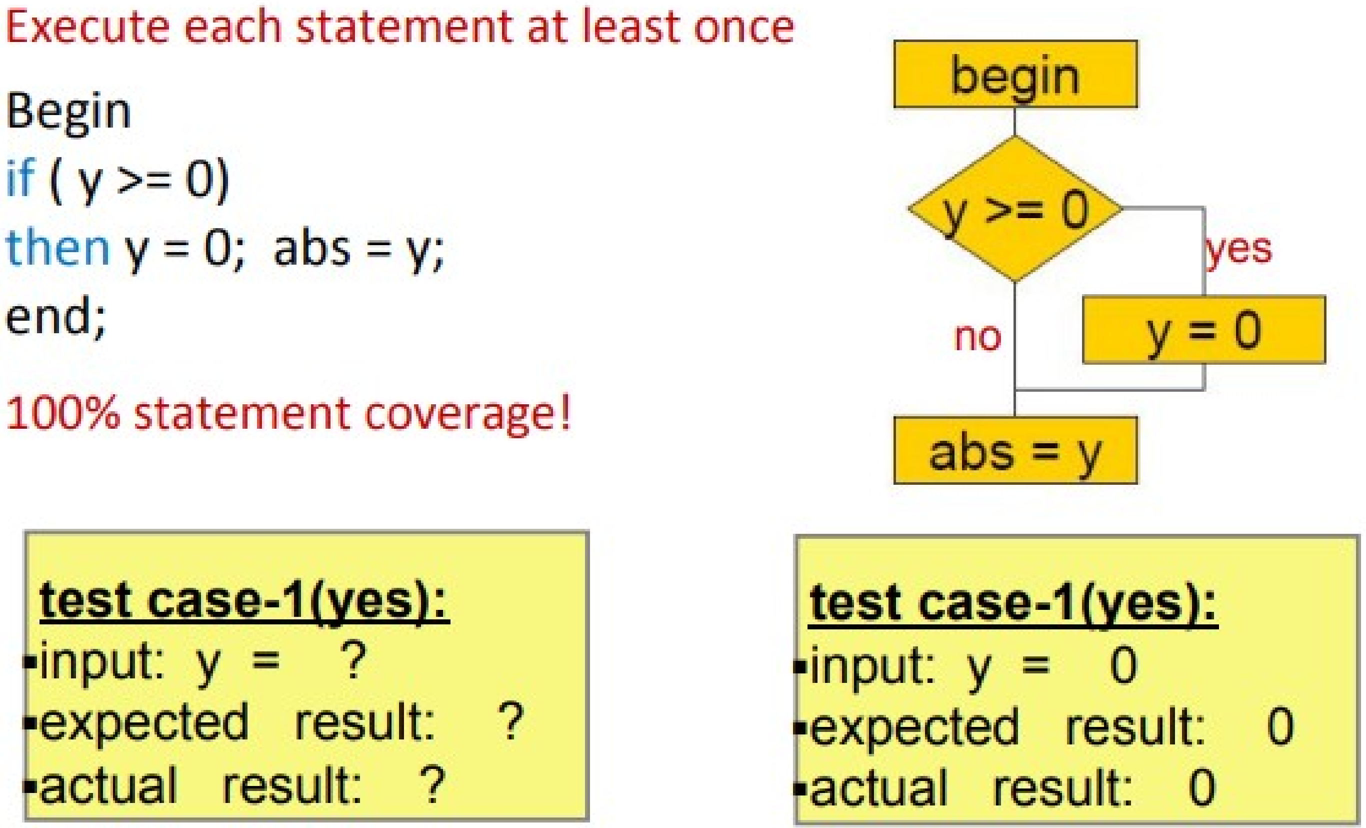
Branch Coverage:

In this scheme, all the possible branches of decision structures are tested. Therefore, sequences of statements following a decision are tested.

Path Coverage:

In path coverage, all possible paths of a program from input instruction to the output instruction are tested. An exhaustive list of test cases is generated and tested against the code.

Statement Coverage



Execute each statement at least once

```
Begin
if ( y >= 0)
then y = 0;  abs = y;
end;
```

100% statement coverage!

test case-1(yes):

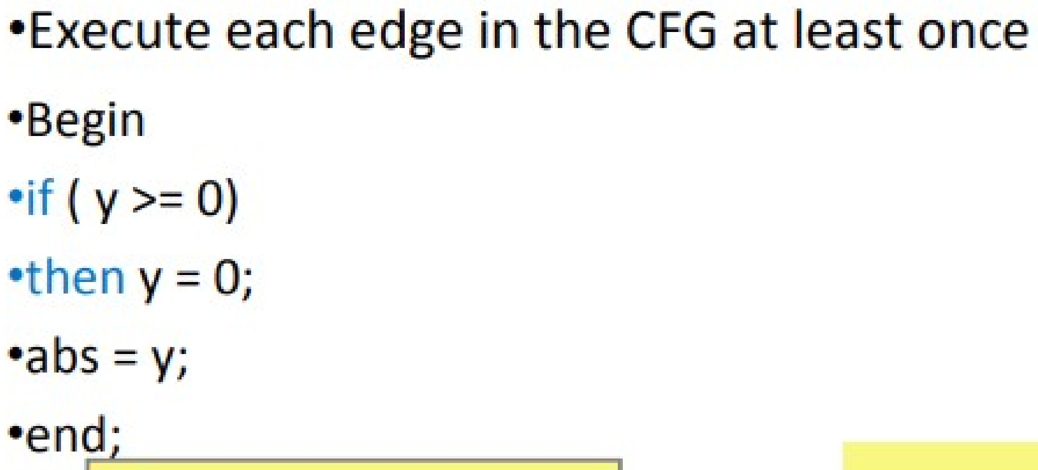
- input: y =   ?
- expected   result: ?
- actual   result: ?

test case-1(yes):

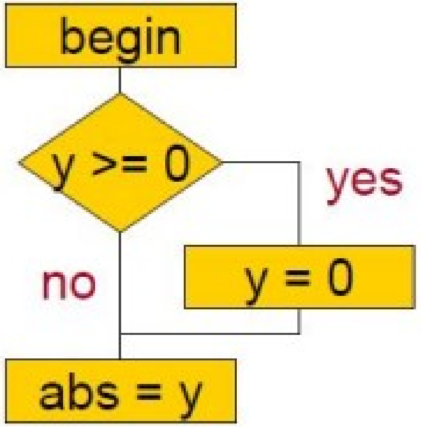
- input: y =   0
- expected   result: 0
- actual   result: 0

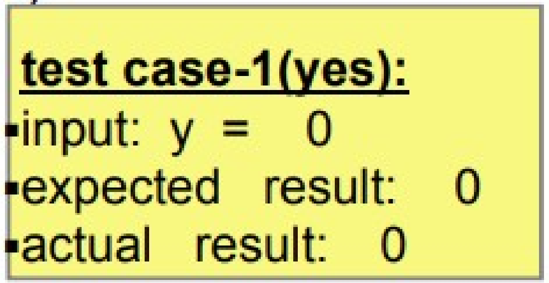
Decision/Branch Coverage

Every statement in the program has been executed at least once, and every decision in the program has taken all possible outcomes at least once.



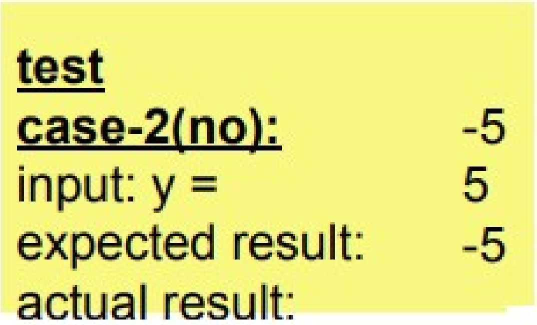
- Execute each edge in the CFG at least once
- Begin
- if (y >= 0)
- then y = 0;
- abs = y;
- end;





test case-1(yes):

- input: y = 0
- expected result: 0
- actual result: 0



test case-2(no): -5

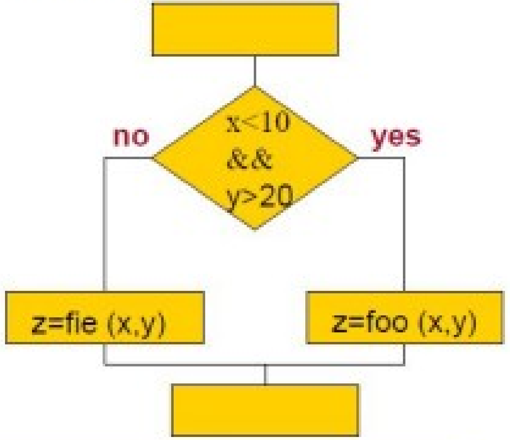
input: y = 5

expected result: -5

actual result:

Decision/Condition Coverage

- Each condition in each decision must be both true and false
- Begin
- if (x < 10 && y > 20) {
- z = foo (x, y);
- else
- z =fie (x, y);
- }
- end;



test case-1(T,F):

- input: x = -4; y =12
- expected result: ?
- actual result: ?

test case-2(F,T):

input: x = 12; y =30

expected result: ?

actual result: ?

Path coverage

- A path is a sequence of branches, or conditions.
- A path corresponds to a test case, or a set of inputs.
- In code coverage testing, branches have more importance than the blocks they connect.
- Bugs are often sensitive to branches and conditions

McCabe's Complexity Metric

Cyclomatic Complexity is a software metric that provides a Quantitative measure of the logical complexity of a program.

McCabe's Complexity Metric

The cyclomatic complexity of the program is computed from its control flow graph (CFG) using the formula:

$$V(G) = Edges – Nodes + 2$$

or by counting the conditional statements and adding 1

This measure determines the basis set of linearly independent paths and tries to measure the complexity of a program.

Cyclomatic Complexity

V(G) = Edges – Nodes + 2

*V(G) = 6 – 6 + 2 = **2***

V(G) = conditional statements + 1

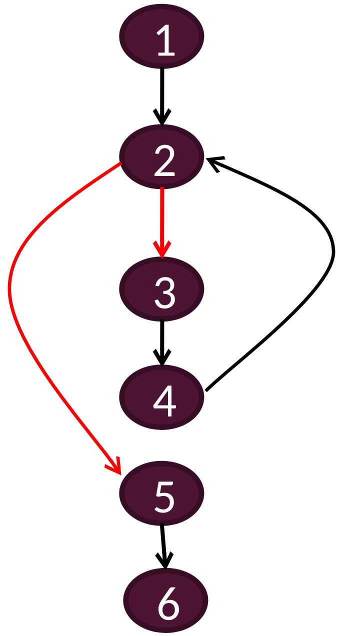
= 1 + 1 = **2**

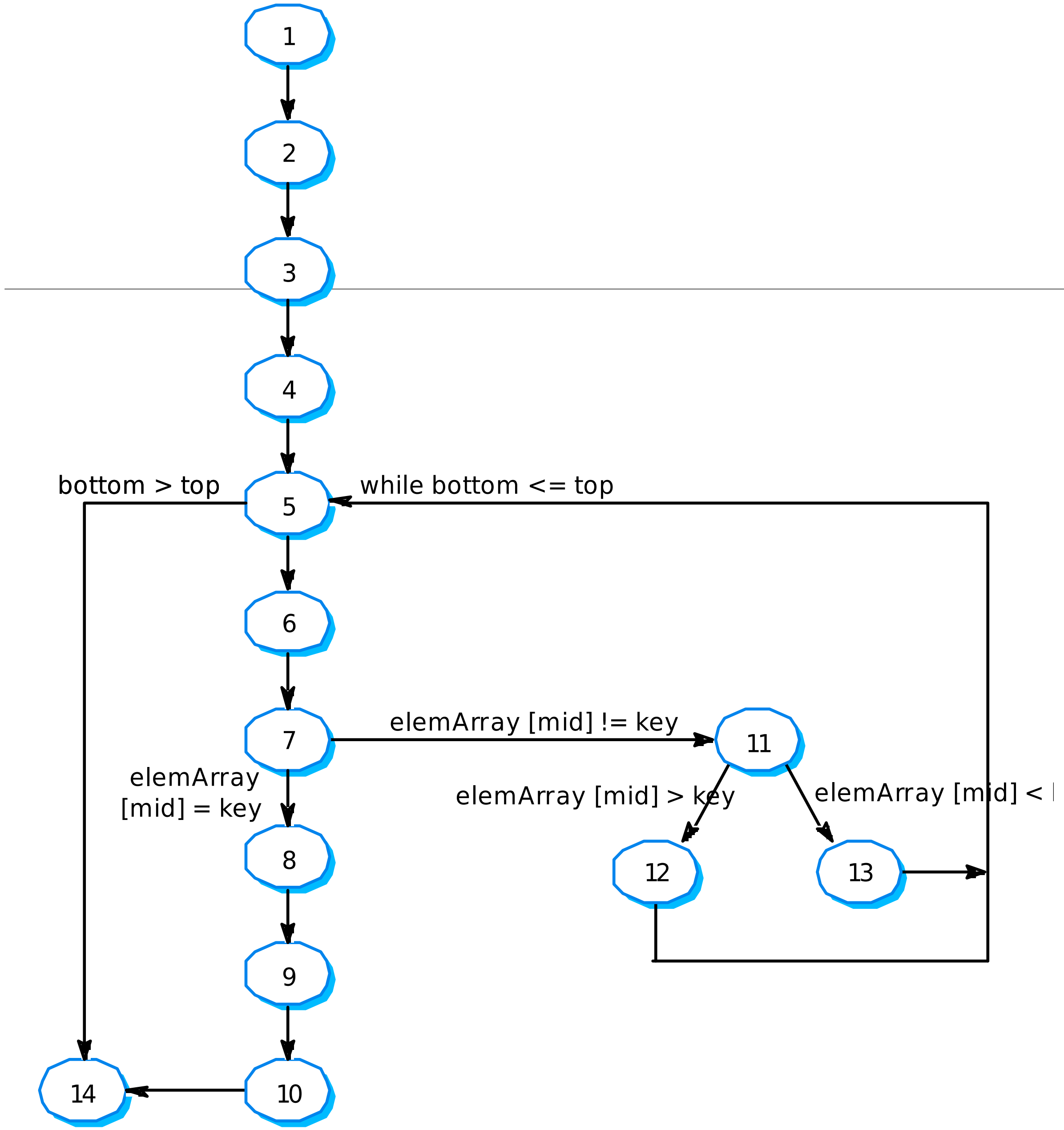
Two linearly independent paths:

1, 2, 5, 6

1, 2, 3, 4, 2, 5, 6

Complexity = 2 < 10 => good quality





V(G) = Edges – Nodes + 2

V(G) = 16 – 14 + 2 = ***4***

V(G) = conditional statements + 1

= 3 + 1 = **4**

four linear independent paths

Complexity = 4 < 10 => good quality

Cyclomatic Complexity

V(G)	Risk
1 – 10	easy program, low risk
11 – 20	complex program, tolerable risk
21 – 50	complex program, high risk
>50	impossible to test, extremely high risk

Applicability and limitation

Control flow testing is the basis of unit testing.

It should be used for all modules of code that cannot be tested sufficiently through reviews and inspections.

Its limitation are that the tester must have sufficient programming skill to understand the code and its control flow

Control flow testing can be very time consuming.

Test Document

Test Document

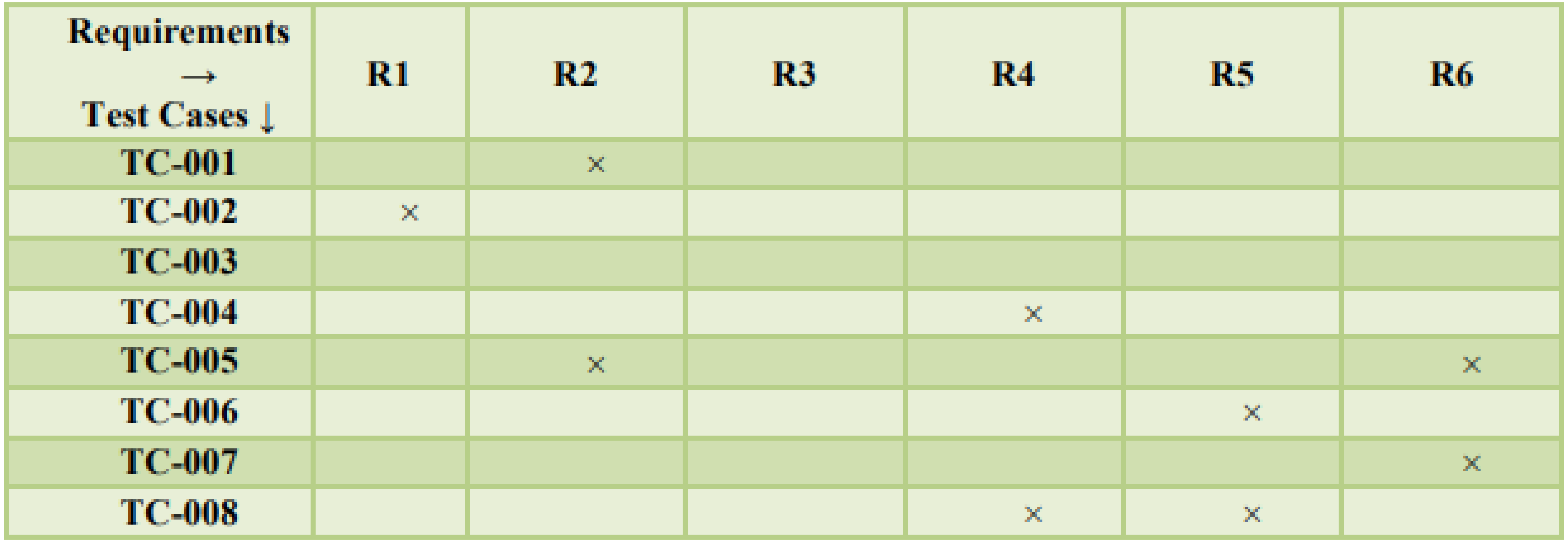
Testing documentation involves the documentation of artifacts which should be developed before or during the testing.

Documentation for Software testing helps in estimating the testing effort required, test coverage, requirement tracking/tracing etc.

This section includes the description of some commonly used documented artifacts related to Software testing such as:

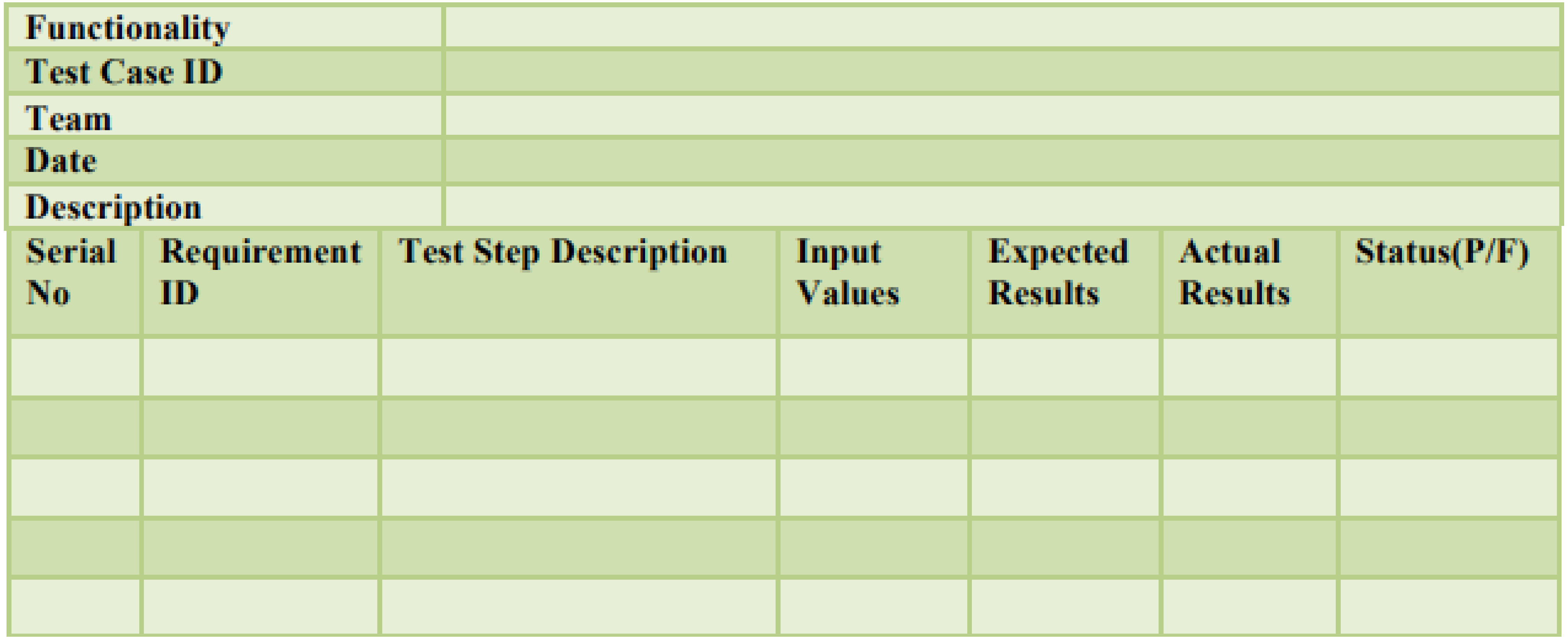
- Test Plan
- Test Scenario
- Test Case
- Traceability Matrix

Requirement Traceability



Requirements → Test Cases ↓	R1	R2	R3	R4	R5	R6
TC-001		×				
TC-002	×					
TC-003						
TC-004				×		
TC-005		×				×
TC-006					×	
TC-007						×
TC-008				×	×	

Test Case Template



Functionality	
Test Case ID	
Team	
Date	
Description	

Serial No	**Requirement ID**	**Test Step Description**	**Input Values**	**Expected Results**	**Actual Results**	**Status(P/F)**

Test Requirements

1. *Functional Requirements*
2. *Non –Functional Requirements*
   - ✓ *Performance*
   - ✓ *Scalability*
   - ✓ *Reliability*

Scenarios for NFRs

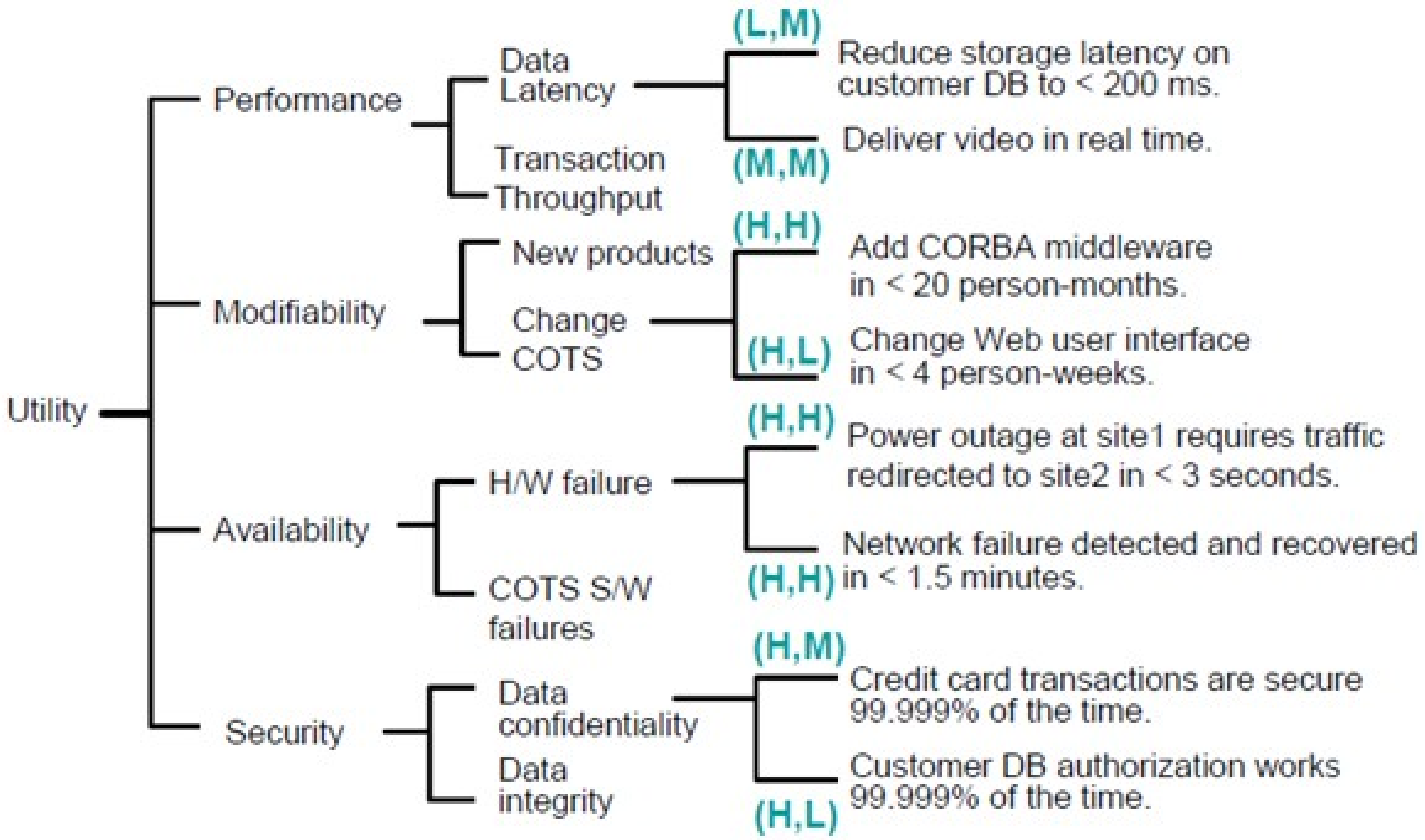
- Scenarios are a technique developed at the SEI to find out issues concerning an architecture through manual evaluation and testing.

- Scenarios are related to architectural concerns such as quality attributes, and they aim to highlight the consequences of the architectural decisions that are encapsulated in the design.

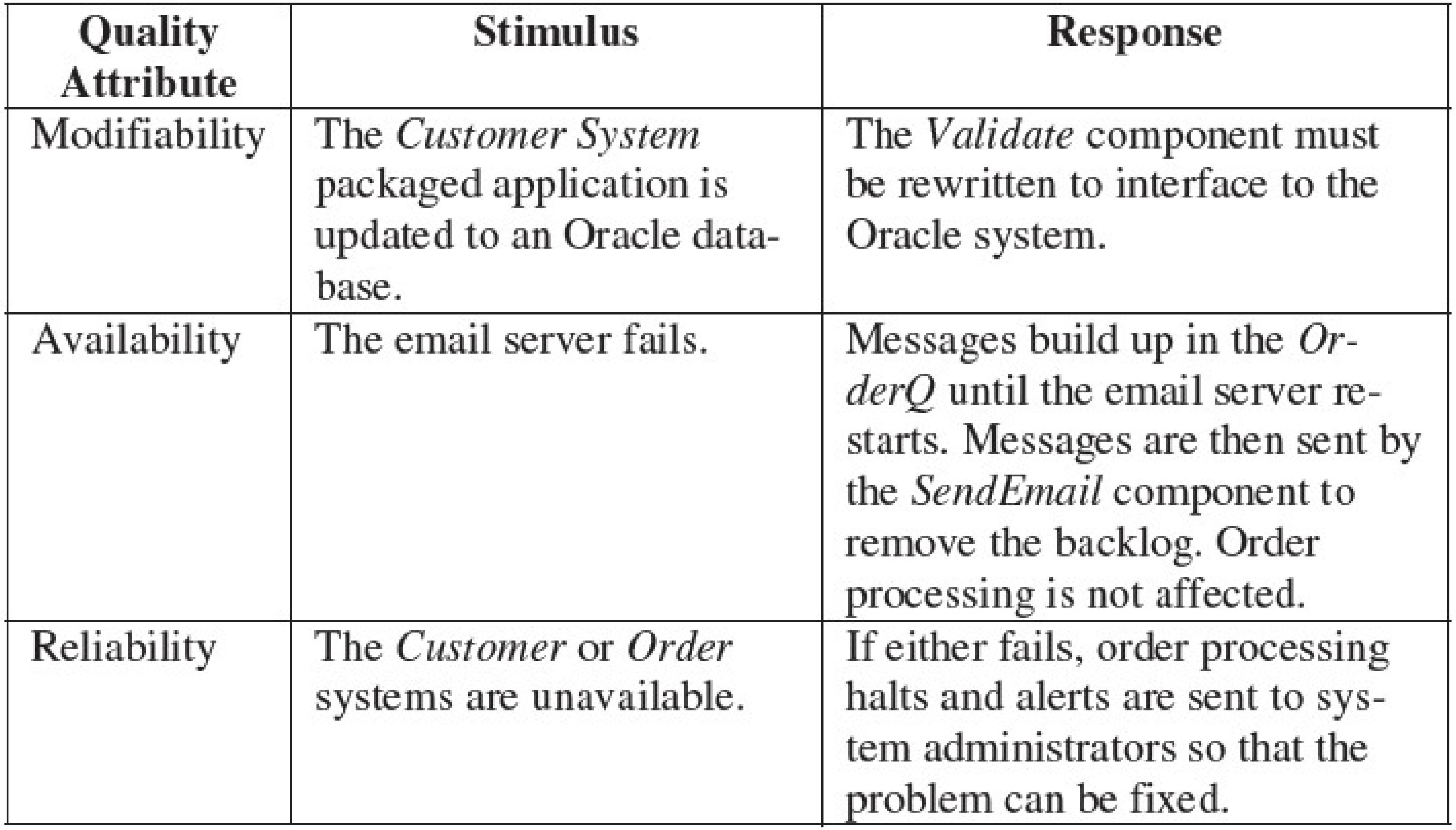
Scenario Types

- **Use case scenarios** reflect the normal state or operation of the system.
- **Growth scenarios** are anticipated changes to the system (e.g., double the message traffic, change message format.
- **Exploratory scenarios** are extreme changes to the system. These changes are not necessarily anticipated or even desirable situations (e.g., message traffic grows 100 times, replace the operating system).

Utility Tree Example


Utility
Performance
Data Latency
Transaction Throughput
(L,M)
Reduce storage latency on customer DB to < 200 ms.
(M,M)
Deliver video in real time.
Modifiability
New products
Change COTS
(H,H)
Add CORBA middleware in < 20 person-months.
(H,L)
Change Web user interface in < 4 person-weeks.
Availability
H/W failure
COTS S/W failures
(H,H)
Power outage at site1 requires traffic redirected to site2 in < 3 seconds.
(H,H)
Network failure detected and recovered in < 1.5 minutes.
Security
Data confidentiality
Data integrity
(H,M)
Credit card transactions are secure 99.999% of the time.
(H,L)
Customer DB authorization works 99.999% of the time.

Utility Tree Example



Quality Attribute	Stimulus	Response
Modifiability	The *Customer System* packaged application is updated to an Oracle database.	The *Validate* component must be rewritten to interface to the Oracle system.
Availability	The email server fails.	Messages build up in the *OrderQ* until the email server restarts. Messages are then sent by the *SendEmail* component to remove the backlog. Order processing is not affected.
Reliability	The *Customer* or *Order* systems are unavailable.	If either fails, order processing halts and alerts are sent to system administrators so that the problem can be fixed.

HAVE A GOOD DAY!