



JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self-Belief | Self Discipline | Self Respect”



**DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING**

**LECTURE NOTES
IT8076 – SOFTWARE TESTING
(Regulation 2017)**

**Year/Semester: III/VI CSE
2020 – 2021**

**Prepared by
Ms. R. Revathi
Assistant Professor/CSE**

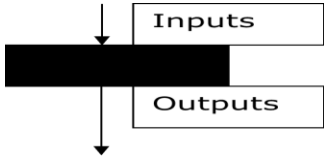
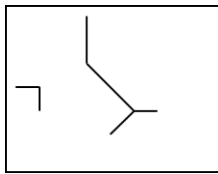
UNIT II TEST CASE DESIGN

Test case Design Strategies – Using Black Box Approach to Test Case Design – Boundary Value Analysis – Equivalence Class Partitioning – State based testing – Cause-effect graphing – Compatibility testing – user documentation testing – domain testing - Random Testing – Requirements based testing – Using White Box Approach to Test design – Test Adequacy Criteria – static testing vs. structural testing – code functional testing – Coverage and Control Flow Graphs –Covering Code Logic – Paths – code complexity testing – Additional White box testing approaches Evaluating Test Adequacy Criteria.

TEST CASE DESIGN STRATEGIES

- Develop effective test cases for execution based testing.
- positive consequences of effective test cases
 - A greater probability of detecting defects
 - A more efficient use of organizational resources
 - A higher probability for test reuse
 - Closer adherence to testing and project schedules and budgets
 - The possibility for delivery of higher quality software products

The two basic testing strategies

Test Strategy & Tester's View	Knowledge Sources	Methods
Black Box 	Requirements Documents Specification Domain Knowledge Defect Analysis Data	Equivalence class Partitioning Boundary value analysis State transition testing Cause and Effect Graphing Error guessing
White Box 	High level Design Detailed Design Control Flow Graphs Cyclomatic Complexity	Statement Testing Branch Testing Path Testing Data Flow testing Mutation Testing Loop Testing

Black Box Testing

- size of the software -> simple module, member function, or object cluster to a subsystem or a complete software system. The description behavior or functionality for the software under test may come from a formal specification

an input/ process output diagram (IPO), or well defined set of pre and post conditions.

- Because the black box approach only considers software behavior and functionality, it is often called functional or specification based testing.
- This approach is useful for revealing requirements and specification defects.

White Box Approach

- Since designing, executing and analyzing the results of white box testing is very time consuming, this strategy is usually applied to smaller sized pieces of software such as module or member function .
- White box testing methods are especially useful for revealing design and code based control, logic and sequence defects, initialization defects and data flow defects.

Using the Black Box Approach to Test Case Design:-

1)Equivalence Class Partitioning:-

Partition the input domain of the software into valid and invalid classes. Invalid classes represent erroneous or unexpected inputs.

Advantage:-

- exhaustive testing - eliminated
- selecting a subset of test inputs with a high probability of detecting a defect
- cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

Guidelines

Input Conditions	no of equivalent classes	EXAMPLE
range of values	one valid & two invalid classes	Eg: range of 1-499 Valid -> all values from 1-99 Invalid -> values < 1 Invalid → values > 499
specific value	one valid & two invalid classes	Eg:- If the specification for a Product code(3115) , Valid -> valid Product code {3115} Invalid -> valid Product code<3115 Invalid → valid Product code>3115
Members of a set	one valid & one invalid classes	eg:- paint module states that the Color RED, BLUE, GREEN and YELLOW are allowed as inputs Valid -> RED Invalid -> BLACK
must be condition (Boolean)	one valid & one invalid classes	Eg:- if a specification for a module states that the first character of a part identifier must be a letter Valid -> TOTAL Invalid -> 3PI

If the input specification in an equivalence class is not handled in an identical way by the software under test, then the class should be further partitioned into smaller equivalence classes

Example: A specification of a square root function.

Function square_root
message (x:real)
when x >0.0
reply (y:real)
where y >0.0 & approximately (y*y,x)
otherwise reply exception imaginary_square_root
end function

input: x (4)
output : y (2) → Square root of x

I) Test Condition Relevant to Input Conditions:

- 1) The input conditions → variable x must be a real number and be equal to or greater than 0.0.
- 2) The output conditions → y must be a real number equal to or greater than 0.0, whose square is approximately equal to x.

II) Generate equivalence classes

- EC1. The input variable x is real, valid.
- EC2. The input variable x is not real, invalid.
- EC3. The value of x is greater than 0.0, valid.
- EC4. The value of x is less than 0.0, invalid.

III) equivalence class reporting Table (EC table)

Condition	Valid EC	Invalid EC
1	EC1 , EC3	EC2 , EC4

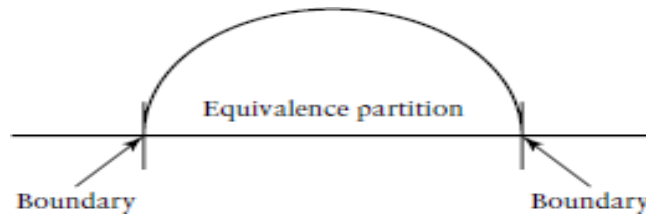
IV) Summary of Test I/ps using EC Partitioning

Test case Id	i/p	Valid EC	Invalid EC
TC1	-3	-	EC4
TC2	4.0	EC1,EC3	-
TC3	AB	-	EC2
TC4	-6.2	-	EC4

Provide testcases for all ECs present in EC table

2)Boundary Value Analysis :

- The test cases developed based on equivalence class partitioning can be strengthened by use of an another technique called boundary value analysis.
- boundary value analysis requires that the tester select elements close to the edges, so that both the upper and lower edges of an equivalence class are covered by test cases.



Guidelines

Input Conditions	Valid & Invalid Test Case	EXAMPLE
range of values	valid test cases → ends of the range, invalid test cases → above and below the end of the range.	Eg: range between -1.0 and +1.0 input values of -1.0, -1.1, and 1.0, 1.1.
number of values	valid test cases → min & Max Numbers Invalid Test Case → Min-1, max+1 numbers	Ex: house can have one to four owners 0,1 owners and 4,5 owners
ordered set,	focus on the first and last elements of the set.	i/p: {25,27,28} last element → 27, 28 ,29 first element → 24, 25 ,26

Example 1:

The input specification for the module states that a widget identifier should consist of 3–15 alphanumeric characters of which the first two must be letters.

I) conditions that apply to the input:

- (i) it must consist of alphanumeric characters,
- (ii) the range for the total number of characters is between 3 and 15, and,
- (iii) the first two characters must be letters.

II) Generate bounds groups

- BLB—a value just below the lower bound
- LB—the value on the lower boundary
- ALB—a value just above the lower boundary
- BUB—a value just below the upper bound
- UB—the value on the upper bound
- AUB—a value just above the upper bound

For our example module the values for the bounds groups are:

- BLB—2 BUB—14
- LB— 3 UB— 15
- ALB—4 AUB—16

Generate Equivalent Classes

Condition 1:

- EC1. Part name is alphanumeric, valid.
- EC2. Part name is not alphanumeric, invalid.

Condition2:

- EC3. The widget identifier has between 3 and 15 characters, valid.
 EC4. The widget identifier has less than 3 characters, invalid.
 EC5. The widget identifier has greater than 15 characters, invalid.

Condition3:

- EC6. The first 2 characters are letters, valid.
 EC7. The first 2 characters are not letters, invalid.

III) Equivalence class reporting table.

Condition	Valid equivalence classes	Invalid equivalence classes
1	EC1	EC2
2	EC3	EC4, EC5
3	EC6	EC7

IV) Summary of Test Inputs using equivalence class & BVA

Test case identifier	Input values	Valid equivalence classes and bounds covered	Invalid equivalence classes and bounds covered
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc ⁺	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6 (typical case)	

Provide testcases for all ECs present in EC table and bound groups.

Example 2: Pin number input of ATM SYSTEM**Case Study : Apply ECP & BVA for pinno of ATM System****Example:**

The Pinno input has following specification

I) Derive Input conditions

- Only digits for pin no input
- Values range from 0000 to 9999 (Length is 4)

II) Generate bounds groups

For our example module the values for the bounds groups are:

BLB— -1 BUB— 9998
LB— 0 UB— 9999
ALB—1 AUB— 10000

Generate Equivalent Classes

Condition1:

EC1. pinno i/p - only digits, valid.

EC2. pinno i/p with digits and other symbols , invalid.

Condition2:

EC3. The pinno i/p has value between 0000 and 9999 , valid.

EC4. The pinno i/p has value < 0000, invalid.

EC5. The pinno i/p has value > 9999, invalid.

III) Equivalence Class Report

i/p condition	valid EC	Invalid EC
1	EC1	EC2
2	EC3	EC4, EC5

IV) Summary of test i/ps

Test Case ID	Input Values	Valid EC & Bounds Covered	InValid EC & Bounds Covered
1.	9999	EC1 , EC3(UB)	-
2.	9998	EC1 , EC3(BUB)	-
3.	0000	EC1 , EC3(LB)	-
4.	0001	EC1 , EC3(ALB)	-
5.	W236	-	EC2
6.	-875	-	EC2 , EC4(BLB)
7.	10000	EC1	EC5(AUB)

3)State based Testing

Graph based testing methods are applicable to generate test cases for state machines such as language translators , work flows , transaction flows and data flows.

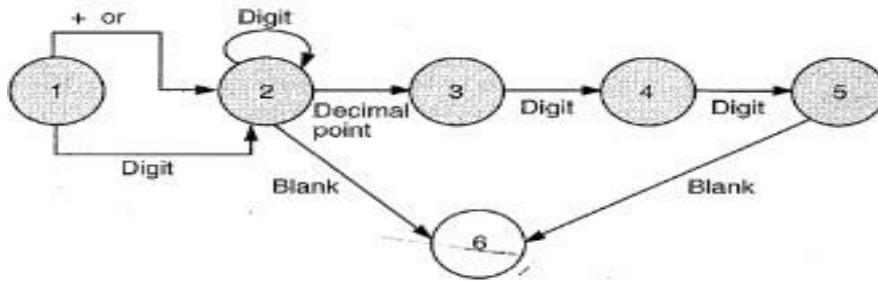
It is useful in

- product is language processor
- work flow modeling
- dataflow modeling

Example: validate number using simple rules (for language processor)

1. number start with an optional sign
2. sign can be followed by nay number of digits
3. digits can be optionally followed by a decimal point, represented by a period
4. if there is a decimal point , then there should be 2 digit after decimal
5. Any Number – whether or not it has a decimal point, should be terminated by a blank.

An example of a state transition diagram.



State transition table

Current state	Input	Next State
1	Digit	2
1	+	2
1	-	2
2	Digit	2
2	Blank	6
2	decimal point	3
3	Digit	4
4	Digit	5
5	Blank	6

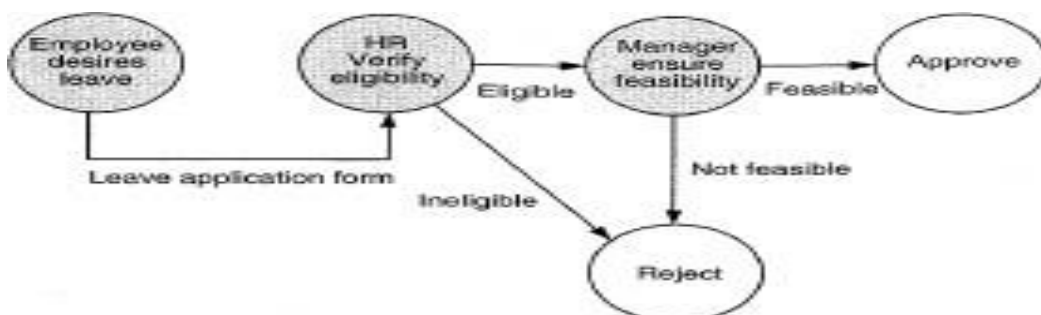
the above state transition table can be used to derive test cases to test valid and invalid numbers

1. Start the start state (state #1)
2. Choose the path that leads to the next state (ex: +/-/digit)
3. Invalid i/p in a given state, generate an error condition TC
4. Repeat the process till u reach the final state

A general outline for using state based testing methods with respect to language processors is

1. Identify the grammar for the scenario. In the above example, we have represented the diagram as a state machine. In some cases, the scenario can be a context-free grammar, which may require a more sophisticated representation of a "state diagram."
2. Design test cases corresponding to each valid state-input combination.
3. Design test cases corresponding to the most common invalid combinations of state-input.

Ex2 : Leave application by an employee (for work flow modeling)



4) Cause effect Graphing

- Equivalence class partitioning does not allow testers to combine conditions .
- It is a dynamic test case writing technique.
- Cause and effect graphing is technique that can be used to combine conditions and derive an effective set of test cases that may inconsistencies in a specifications
- It restates the requirements specification in terms of logical relationship between the input and output conditions. Since it is logical, it is obvious to use Boolean operators like AND, OR and NOT.

Steps:

1. The tester must decompose the specification of a complex software component into lower level units

2. Identify causes and effects

Cause - distinct i/p condition or an equivalence class of i/p conditions.

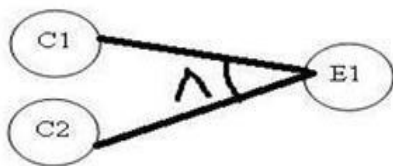
Effect - an output condition or a system transformation

3. From the cause and effect information, a Boolean cause and Effect graph is created.

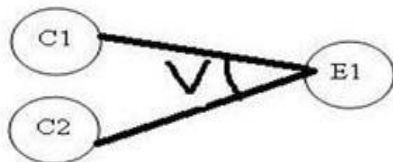
Graph : Node \rightarrow causes(Left Side) and effects (Right side). logical operators such as AND, OR and NOT and are associated with the arcs.

Notations for constructing cause and Effect graph

AND – For effect E1 to be true, both the causes C1 and C2 should be true



OR – For effect E1 to be true, either of causes C1 OR C2 should be true



NOT – For Effect E1 to be True, Cause C1 should be false



4. The graph may be annotated with constraints that describes combinations of causes and/or effects that are not possible due to environmental or syntactic constraints
5. Convert the graph into a decision table.
6. The columns in the decision table are transformed into test cases.

Example: module that allows user to perform a search for a character in an existing string.

Step1 : decompose the specification

Input \rightarrow length of the string
character to search for.

Output → Char position
 NOT FOUND
 out of range

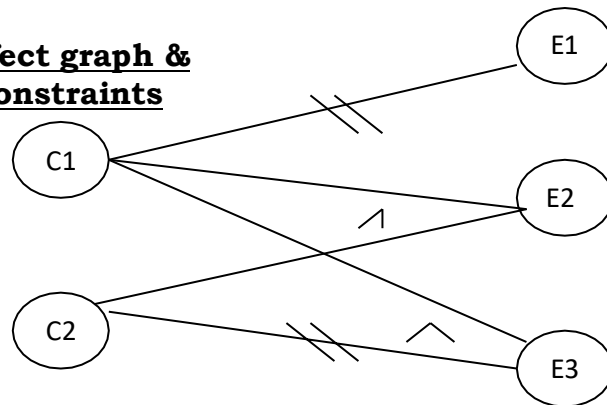
Step2 : Identify causes and effects

- C1 : Positive integer from 1 to 80
- C2 : Character to search for is in String
- E1 : Integer out of range
- E2 : Position of character in string
- E3 : Character not found.

Rules or relationship :-

- If C1 and C2, then E2.
- If C1 and Not C2, then E3
- If not C1, then E1.

Step 3: Construct cause and Effect graph & Step: 4 Graph annotated with constraints



Step 5: Convert the graph into a decision table(1- true , 0-false , - don't care)

	T1	T2	T3
C1	1	1	0
C2	1	0	-
E1	0	0	1
E2	1	0	0
E3	0	1	0

Step6 : Decision table are transformed into test cases

Columns are changed into testcases

Existing string → “abcde”

Test Cases	Length	Character to search for	Outputs
T1	5	C	3
T2	5	w	Not Found
T3	90		Integer out of range

5)Compatibility testing

- It ensures the working of the product with different infrastructure components (Non-functional testing).
- test case results depend on infrastructure for delivering functionality

- infrastructure parameter are changed , product is expected to still behave correctly and produce desired results.
- infrastructure parameter → H/W , S/W , other components
Example
- **Test the application in same browsers but in different versions.** For e.g. to test the compatibility of site ebay.com. Download different versions of Firefox and install them one by one and test the ebay site. Ebay site should behave equally same in each version.
- **Test the application in different browsers but in different versions.** For e.g. testing of site ebay.com in different available browsers like Firefox, Safari, Chrome, Internet Explorer and Opera etc.

Parameters:

- Processor (Pentium III / IV, Xenon, SPARC)
- Architecture(32 bit / 64 bit)
- Resource Availability (RAM & Hard disk space)
- Equipment (printer , Modem, Router).
- Operating System
- Middle-tier infrastructure components (Web Server, App server)
- Back end components (Oracle, MS SQL)
- Any s/w used to generate product binaries (compiler, linker)
- Technological components (SDK, JDK)

Compatibility matrix :

Each row represents a unique combination of a specific set of values of the parameter

Ex: Mail App

Server	App Server	Web Server	Client	Browser	MS Office	Mail Server
Windows 2000 Microsoft SQL server 2000	Windows 2000 Advanced server with SP\$ and .Net framework 1.1	IIS5.0	Win2K Professional	IE 6.0	Office 2k & Office XP	Exchange 5.5 & 2K
					

Common Techniques

- Horizontal Combination(HC): Parameters of the row grouped together for executing the test cases
- Intelligent Sampling:
 - In HC each feature of the product has to be tested with each row in the compatibility matrix→ involves time & effort
 - Various permutation and combination methods used
 - Selection of intelligent sampling based on

- Information collected on the set of dependencies of the product with parameter.
- Less dependent → removed from the list
- Can include parameters that are part of product

Types:

- **Backward compatibility Testing** is to verify the behavior of the developed hardware/software with the older versions of the hardware/software. The product parameters required for backward compatibility is added to the compatibility matrix and are tested.
- **Forward compatibility Testing** is to verify the behavior of the developed hardware/software with the newer versions of the hardware/software.

Tools for compatibility testing:

- Adobe Browser Lab - Browser Compatibility Testing
- Secure Platform - Hardware Compatibility tool
- Virtual Desktops - Operating System Compatibility

6) User documentation testing:

User documentation testing Is done to ensure the documentation matches the product and vice versa.

User Documentation includes

Manuals	user guides	installation guides	setup guides
online help	read me files	software release notes	

Objective

- To check if what is stated in the document is available in the product
- To check if what is there in the product is explained correctly in the document.

- Product upgraded → documentation upgraded
- Lack of coordination → documentation group & testing /development group
- sitting in front of the system & verifying screen by screen , transaction by transaction , report by report
- checks language aspects (spell check & grammar)

Advantages

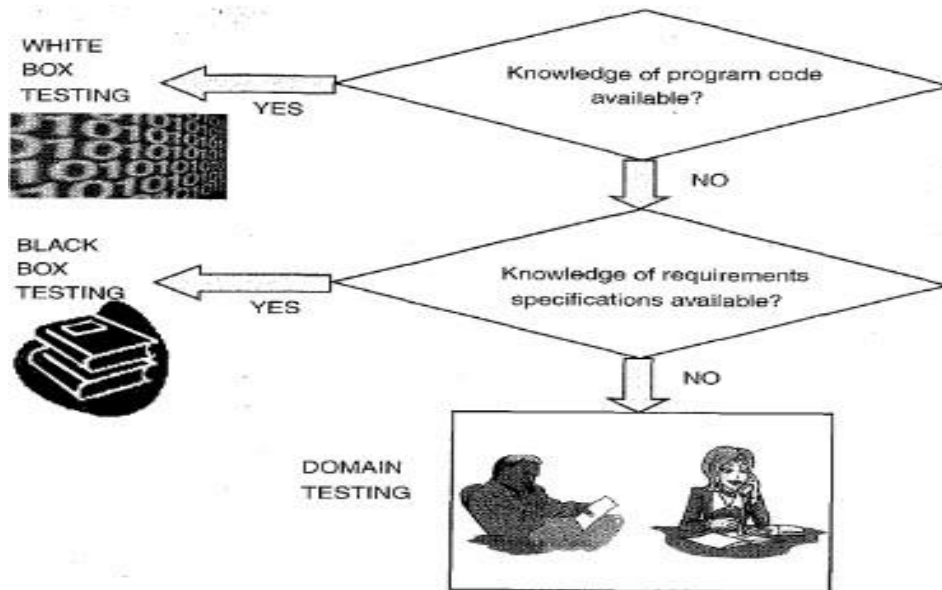
- Aids in highlighting problems overlooked during reviews
- High quality documentation minimizes defect reported by the customer
- Results in less difficult support calls
- New Programmer & testers can use doc. to learn the external functionality of the product
- Customer need less training & can proceed more quickly to the advanced training

The effort & money spent on this would form a valuable investment in the long run for the organization.

7) Domain Testing :

- Testing the product purely based on domain knowledge & expertise in the domain of application
- Requires business domain knowledge, Extension of black box testing

When to apply domain testing?



- Ability to design and execute test cases that relate to the people who will buy and use the software.
- Concerned about everything in the business flow
- Testing the product , not by going through the logic built into the product.
- Business flow determines the steps, not the software under test → “ Business Vertical Testing”
- To Test the software for “Domain Intelligence” , tester is expected to have intelligence & knowledge of business flow
- Earlier phases of Black box Testing deals with Equivalent Class Partitioning ,Decision Table (Cause Effect Graphing)
- Domain testing is done all components are integrated and product has been tested using black box approaches.

Ex: Cash Withdrawal of ATM system

- Step1** :Go to the ATM
- Step2** : Put ATM card inside
- Step3** : Enter Correct PIN
- Step4** : Choose cash withdrawal
- Step5** : Take the cash
- Step6** : Exit and retrieve the card

Other Black Box testing →Required denomination is available to dispense the requested amount

Domain Testing →Whether user has got the right amount / not

8) Random Testing:-

If a tester randomly selects input from the domain, this is called random testing

- Eg:- if the valid input domain for a module is all positive integer between 1 and 100,
- would randomly or unsystematically select valued form within that domain; for example the values 55,24,3 might be chosen

9) Requirements Based Testing

- Deals with validating the requirements given in the SRS
- Requirements → 1) Explicit 2) Implicit
- Precondition
 - Detailed review of the requirements specification, it ensures that they are consistent , correct , complete , testable.
 - Implied requirements are converted and documented as explicit requirements → more effective
- Explicit & Implicit requirements are collected and documented as “ Test Requirements specification “ (TRS)
- Requirements are tracked by a Requirement Traceability Matrix (RTM)
- RTM traces all the requirements from their origin through design, development and testing.

Example : Locking (key is turned clockwise)
unlocking (key is turned anticlockwise)
Key No :123-456

Sample Requirement Specification

Sno	ReqId	Description	Priority (H, M,L)
1.	BR-01	Inserting the key numbered 123 -456 and turning it clockwise should facilitate locking	H
2.	BR-02	Inserting the key numbered 123 -456 and turning it anticlockwise should facilitate unlocking	H
3.	BR-03	only key number 123-456 can be used to lock and unlock	H
4.	BR-04	No other object can be used to lock	M
5.		

Requirement Traceability Matrix (RTM)

Req Id	Description	Priorty (H, M,L)	Test Conditions	Test Case IDs	Phases of testing
BR-01	Inserting the key numbered 123 -456 and turning it clockwise should facilitate locking	H	Use key 123-456	Lock_001	unit Component

BR-02	Inserting the key numbered 123 -456 and turning it anticlockwise should facilitate unlocking	H	Use key 123-456	Lock_002	unit , Component
BR-03	only key number 123-456 can be used to lock and unlock	H	Use key 123-456 to lock	Lock_003	Component
			Use key 123-456 to unlock	Lock_004	
				

- Tests for higher priority requirements will get precedence over tests for lower priority → functionality that has higher risk is tested earlier
- cross ref b/w requirements and the subsequent phases is recorded in the RTM
- RTM helps in identifying the relationship between the requirements and test cases. Combinations are
 - one (requirements) to one (Test Case)
 - one to many
 - many to one
 - many to many
 - one to none.

RTM in Requirement Based Testing:

- it is a tool to track the testing status of each requirement, without missing any requirements
- prioritization enables selecting the right features to test
- list of test cases that address the particular requirement can be viewed

Test metrics

1. Requirements addressed priority wise
2. Number of test case requirement wise
3. Total no of test cases

Test results

1. Total no of test cases passed
2. Total no of test cases failed
3. Total no of defects in requirements
4. No of requirements completed
5. No of requirements pending

Summary of Test I/P:

s.no	Req Id	Priority	Test Cases	Total test cases	Total test cases Passed	Total test cases Failed	% Pass	No of defects
1.	BR-01	H	Lock_01	1	1	0	100	1
2.	BR-02	H	Lock_02	1	1	0	100	1
3.	BR-03	H	Lock_03,04	2	1	1	50	3
4.				...				

Using above Observations can be made with respect to the requirement

10) Positive & Negative Testing

Positive testing

- Verifies the requirements of the product & set of expected o/p
- To prove that the product work as per specification
- is to prove that the product works as per specification and expectations. A product delivering an error when it is expected to give an error, is also a part of positive testing.
- +ve testing is done to verify the known test conditions.

Ex; Lock & Key

Req. no.	Input 1	Input 2	Current state	Expected output
BR-01	Key 123-456	Turn clockwise	Unlocked	Locked
BR-01	Key 123-456	Turn clockwise	Locked	No change
BR-02	Key 123-456	Turn anticlockwise	Unlocked	No change
BR-02	Key 123-456	Turn anticlockwise	Locked	Unlock

Negative Testing

- Negative testing is done to show that the product does not fail when an unexpected input is given. The purpose of negative testing is to try to break the system. Negative testing covers scenarios for which the product is not designed and coded. In other words, the input values may not have been represented in the specification of the product.
- -ve testing is done to break the product with unknowns

S. No. ✓	Input 1	Input 2	Current state	Expected output
1	Some other lock's key	Turn clockwise	Lock	Lock
2	Some other lock's key	Turn anticlockwise	Unlock	Unlock
3	Thin piece of wire	Turn anticlockwise	Unlock	Unlock
4	Hit with a stone		Lock	Lock

Difference between positive testing and negative testing

For positive testing if all documented requirements and test conditions are covered, then coverage can be considered to be 100 percent.	no end to negative testing, and 100 percent coverage -ve testing is impractical. Negative testing requires a high degree of among the testers to cover as many "unknowns" as possible to avoid at a customer site.
---	---

Summary of Black Box Testing

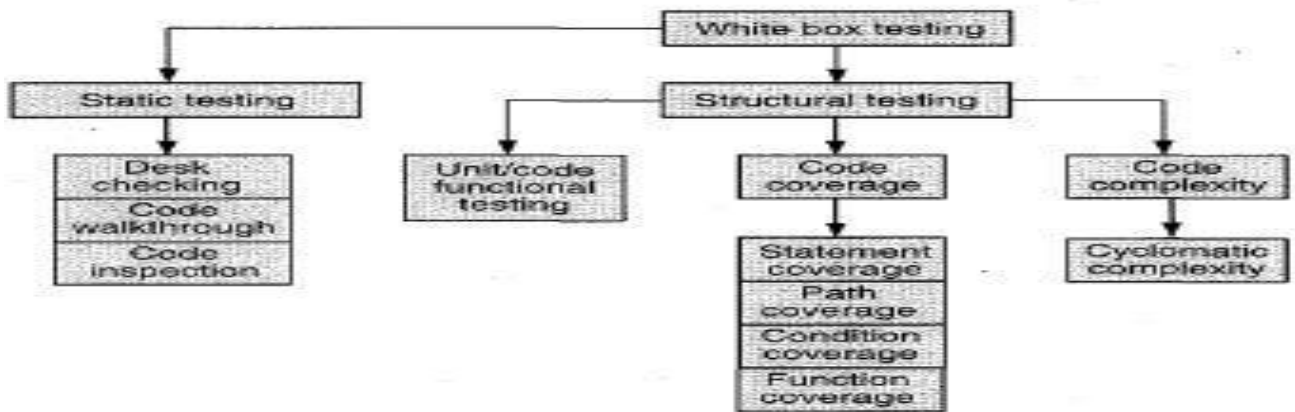
i/p values divided into class	1. Equivalent Class Partitioning
i/p values in range	2. Boundary Value Analysis
i/p & o/p values with multiple	3. Cause effect graphing

Condition	
Checking expected & un expected i/p Values	4. Positive & negative Testing
Language processor , work flow ,process flow	5. State based testing
To ensure the Requirements	6. Requirement Based Testing
To test domain expertise	7. Domain Testing
Documentation consistent with Product	8. Documentation Testing

Using the White Box Approach to Test Case Design

- white box Testing The tester’s goal is to determine if all the logical and data elements in the software unit are functioning properly.
- during the detailed design phase of development - knowledge needed for the white box test design approach often becomes available to the tester in later phase of software life cycle
- White Box test design follows black box design as the test efforts for a given project progress in time

White Box	Black Box
white box based test design is most useful when testing small components.	Black box useful for both small & large s/w components
Level of detail required for test design is very high	Comparatively low



Static Vs Structural Testing

Static	Structural
Product is tested by tester by going through the source code not the executable or binaries	Tests are run by computer on the built product
Does not involve executing the Program	Involves executing the program against predesigned test cases

Static test reveals :

- Code works according to the functional requirements

- Code has been written in accordance with the design developed earlier in the project life cycle
- Code for any functionality has been missed out
- Code handles errors properly

Static test Methods :

1. Desk Checking of the code

- Informal checking done by author
- No structured method
- No Logs / check lists
- Depends on knowledge of the author

Disadvantages

- A developer is not the best person to detect problems in his or her own code. He or she may be tunnel vision and have blind spots to certain types of problems.
- Developers generally prefer to write new code rather than do any form of testing
- This method is essentially person-dependent and informal and thus may not work consistently across all developers.

2. Code walkthrough

- **Group oriented** - method and formal inspection are group-oriented methods
- **Multiple perspective** – walkthroughs and inspections is very thin and varies from organization to organization. The advantage is that it brings multiple perspectives
- **Multiple roles** - a set of people look at the' program code and raise questions for the author. The author explains the logic of the code, and answers the questions. If the author is unable to answer some questions, he or she then takes those questions and finds their answers

3. Formal /Fagan Inspection:

- 1) Group oriented , highly formal & structured
- 2) specific roles , requires thorough preparation

This method increases the number of defects detected by

- 1) demanding thorough preparation before an inspection/review;
- 2) enlisting multiple diverse views;
- 3) assigning specific roles to the multiple participants; and
- 4) going sequentially through the code in a structured manner.

Roles :

- 1) Author- programmer or developer
- 2) Moderator - expected to formally run the inspection according to the process.
- 3) Inspector/Reviewer - provides, review comments for the code.

- 4) Scribe - takes detailed notes during the inspection meeting and circulates them to the inspection team after the meeting.

Process:

Author & Moderator select the review team

Introductory Meeting

- o Author present his perspective
- o Typical Document (code, Design, SRS, Stds) is circulated
- o Moderator informs date ,time venue – inspection meeting

Defect Logging Meeting:

The moderator takes the team sequentially through the program code, asking each inspector if there are any defects in that part of the code. If any of the inspectors raises a defect, then the inspection team deliberates on the defect and, when agreed that there is a defect, classifies it in two dimensions

1)Major (major defects need immediate attention.) / **Minor** – (may not substantially affect a program)

2) **Systematic** (machine-specific idiosyncrasies may have to removed by changing the coding standards) / **mis execution**(happens because of an error or slip on the part of the author. example : a wrong variable in a statement)_

Review Meeting (if the defect severe)

Challenges in Formal /Fagan Inspection

- Time Consuming
- Logistics & Scheduling - multiple people involved
- Not possible to review entire coding

Based on criticality & complexity of code is classified into

High ,medium	Formal Inspection
Low	Walk through , desk checking

Structural testing Methods:

The fundamental difference between structural testing and static testing is that in structural testing tests are actually run by the computer on the built product, whereas in static testing, the product is tested by humans using just the source code and not the executables or binaries.

1. Unit Functional Testing – methods fall under debugging category

- a. **Initially Quick test** – the developer can perform certain obvious tests, knowing the input variables and the corresponding expected output Variables
- b. **modules with Complex logic & condition** – build debug version(ex: intermediate print statement)
- c. **run the product under debugger or IDE** (single stepping of instruction, break points etc)

2. Code Coverage Testing

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing.

instrumentation

- % of code covered by testing is found by a technique
- specialized tool to rebuild the product , link with the set of lib
- Reporting on the portion of the code covered frequently, so easy to identify critical & most often code.

Types of coverage

- a)Statement Coverage
- b)Path Coverage
- c)Condition Coverage
- d)Function Coverage

a)Statement Coverage

It refers to writing test cases that execute each of the program statements.

There are 4 types of programming constructs.

1. **Sequential control flow**
2. **Two-way decision statements like if then else**
3. **Multi-way decision statements like Switch**
4. **Loops like while do, repeat until and for**

1)Sequential Control flow(SC)

- Generate test data to make the program enter the sequential block, to make it go through the entire block
- this may not always be true , Asynchronous Exceptions - (for example, a divide by zero)
- Multiple Entry Point , in Non Structured Programming

SC metric= No of of statements exercised / Total No of Statements

2) Two-way decision statements -if then else

- Have data to test the then part
- Have data to test the else part
- Relevance of statement coverage ?

If the program implements wrong requirements and this wrongly implemented code is "fully tested," with 100 percent code coverage, it still is a wrong program and hence the 100 percent code coverage does not mean anything.

Ex:

```
Total =0;
If(code =='M')
{ Stm1;
...
  Stm7; }
Else
```

Percent = value/Total *100; /*divide be zero*/

when we test with code = "M," we will get 80 percent code coverage. But if the data distribution in the real world is such that 90 percent of the time, the value of code is not = "M," then, the program will fail 90 percent of the time (because of the divide by zero in the highlighted line).

3) multi way decision statements – switch

It can be reduced to multiple two way if statement

4) Loops – while do ,repeat until , for

Looping statements can be handled in 3 ways.

- 1) Skip the loop
 - so that the situation of the termination condition being true before starting the loop is tested.
- 2) Exercise the loop between one & max number of times
 - to check all possible "normal" operations of the loop
- 3) Cover the loop around the boundary (i.e n-1, n,n+1)

b)Path Coverage

statement coverage may not indicate “true coverage”. path coverage gives better representation, split a program into a number distinct paths. A program can start from the beginning and take any of the paths to its completion.

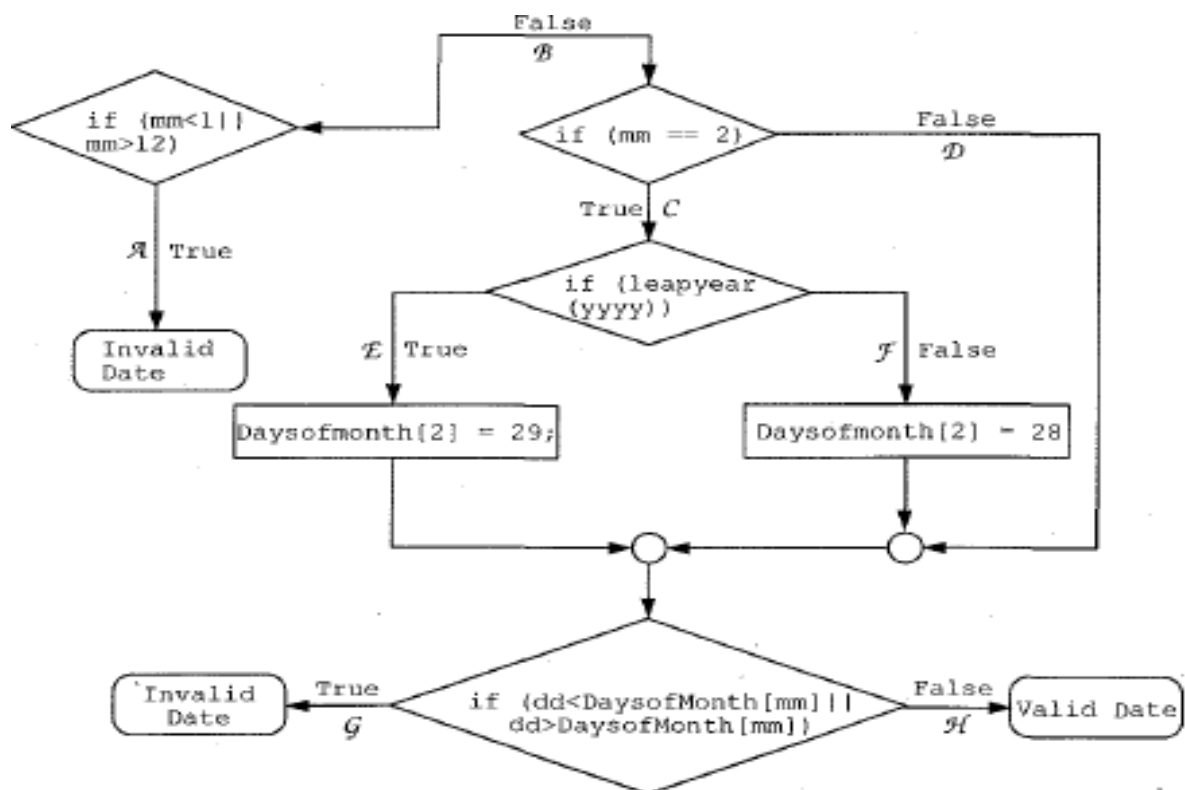
$$\text{Path Coverage} = \frac{\text{No of of path exercised}}{\text{Total No of of path in the program}}$$

Ex: Date validation routine . date accepted as 3 fields dd, mm, yyyy.

i/p → validate numeric i/p for date (dd , mm , yyyy)

leapyear() function (returns FALSE/ TRUE based on i/p)

array → dayofMonth[] contains No of days in each month



The Flow chart shows different Path can be taken through program. Path label is given as A... H .

A	B-D-G B-D-H	B-C-E-G B-C-E-H	B-C-F-G B-C-F-H
If Wrong Month given	Except Feb Month with correct and wrong date given	Feb Month & Leap year with correct and wrong date given	Feb Month & Not Leap year with correct and wrong date given

Summary of Test inputs

TC ID	Path (Description)	Input	Expected o/p
TC1	A (Month Wrong Path)	20/ 0/2000	Invalid Date
TC2	B-D-G (Not Feb - days wrong)	31/4/2015	Invalid Date
TC3	B-D-H (Not Feb - days correct)	31/1/2015	valid Date
TC4	B-C-E-G (Feb , Leap Year - days wrong)	30/2/2016	Invalid Date
TC5	B-C-E-H (Feb , Leap Year - days correct)	29/2/2016	valid Date
TC6	B-C-F-G (Feb , Not Leap Year - days wrong)	29/2/2014	Invalid Date
TC7	B-C-F-H (Feb , Not Leap Year - days Correct)	10/2/2014	valid Date

C)Condition Coverage

- It is necessary to have test cases that exercise each Boolean expression and have test cases test produce the TRUE and FALSE paths.
- Further refinement of path coverage , Make sure each Boolean expression is covered for TRUE as well as FALSE paths
- Ex: Path A covered on giving $mm < 1$, reporting invalid month
Program not tested for $mm > 12$
- Compilers perform optimizations to minimize the number of Boolean operations and all the conditions may not get evaluated, even though the right path is chosen.
- For example, when there is an OR condition (as in the first IF statement above), once the first part of the IF (for example, $mm < 1$) is found to be true, the second part will not be evaluated at all as the overall value of the Boolean is TRUE.
- Similarly, when there is an AND condition in a Boolean expression, when the first condition evaluates to FALSE, the rest of the expression need not be evaluated at all.
- For all these reasons path testing is not sufficient.

<p>Condition Coverage= No of of conditions exercised/Total No of of conditions in the program Above formula indicates percentage of conditions covered by a set of test cases.</p>
--

d)Function Coverage

- This testing finds how many functions are covered by test cases
- Ex: Database s/w -- inserting a row into the database
Payroll app – calculate tax
- **Adv:**
 - 1) Functions are easier to identify
 - 2) Higher level of abstraction than code, easy to achieve 100%

- 3) more logical mapping to requirements than other type of coverage
- 4) importance of functions can be prioritized based on the importance of requirements
- 5) provides natural transition to black box testing

Function Coverage = No of of function exercised/Total No of of functions in the Program

3)Code Complexity Testing

While using these coverage : following questions are raised

- 1) which of the paths are independent ?(to minimize the test cases)
- 2) is there an upper bound on the number of tests to be executed to ensure all the statements have been executed at least once ?

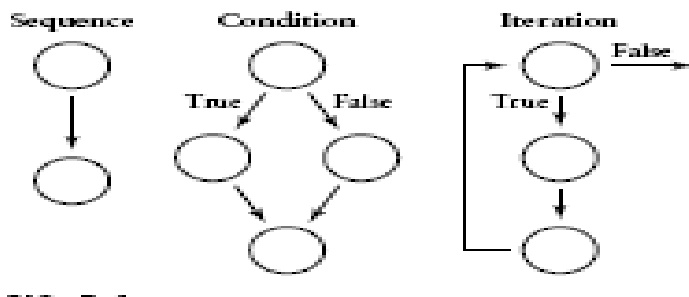
Ans : Cyclomatic complexity “ a metric that quantifies the complexity of a program”

Steps in Determining Cyclomatic complexity

- 1) Construct Flow Graph
- 2) Compute cyclomatic complexity of the resultant flow graph
- 3) Determine a basis set of linearly independent paths
- 4) Prepare test cases that will force execution of each path in the basis set

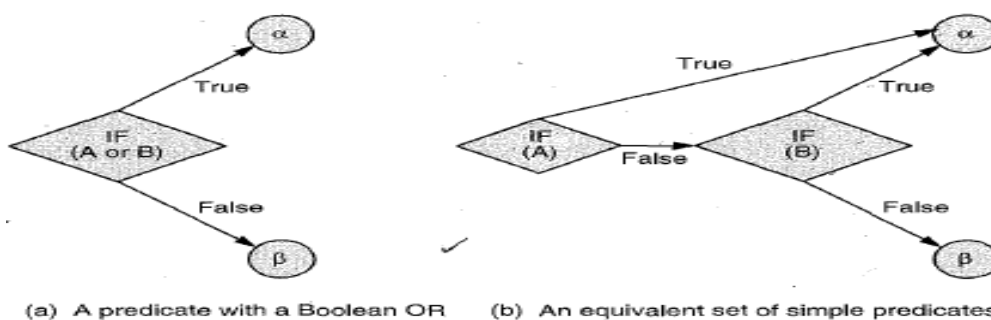
Flow Graph

- Program is represented in the form of a flow graph.
- Flow graph can be constructed like a flowchart.
- Flow graph consist of nodes and graphs.
- Representation of Programming primitives in flowgraph

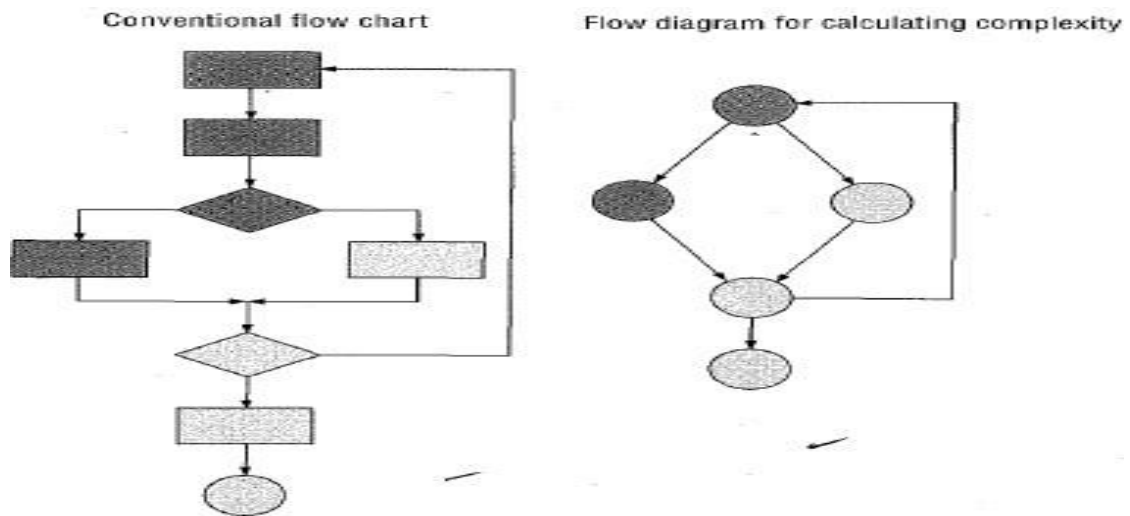


Steps to convert flowchart into flow graph

- 1) Identify the predicates or decision points
- 2) Ensure that the predicates are simple



- 3) Combine all sequential statements into a single node
- 4) When a set of sequential statements are followed by a simple predicate , combine all the sequential statements & predicate check into one node & have 2 edges emanating from this one node
- 5) Make sure that all the edges terminate at some node.



To Compute cyclomatic complexity : 3 ways

- i) Cyclomatic Complexity $V(G) = E - N + 2$, where E is the number of edges and N is the number of nodes in graph G
- ii) $V(G) = P + 1$, where P is the number of predicate nodes in the flow graph G
- iii) $V(G)$ =the number of regions (Closed & Outer Region)

Example : sum of all positive numbers (greater than zero)

a → array name

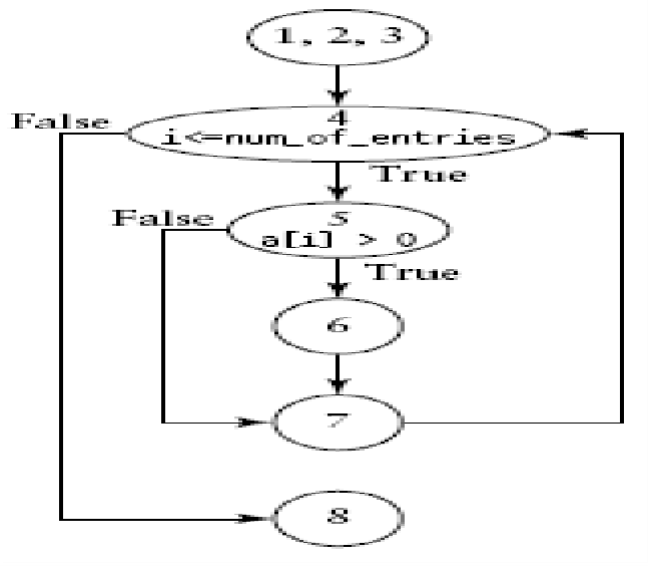
num_of_entries → no of elements

sum → to store total value

1. pos_sum(a, num_of_entries, sum)
2. sum=0
3. int i=1
4. while (i <=num_of_entries)
5. if (a[i] >0)
6. sum=sum+a[i]
7. endif
8. i=i+1
9. end while
10. end pos_sum

Assign line no for each statement in the program before constructing the flowgraph.

1) Construct Flow Graph



2) Calculate the cyclomatic complexity of the resultant flow graph

- i) $V(G) = E - N + 2$ $E = 7, N = 6$ $V(G) = 7 - 6 + 2 = 3$
- ii) $V(G) = P + 1$ $P = 2$ $V(G) = 2 + 1 = 3$
- iii) $V(G) = \text{No of Regions}(R)$ $R = 3$ $V(G) = 3$

3) Determine a basis set (independent path)

A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.

- (i) 1-2-3-4-8 (skip the loop)
- (ii) 1-2-3-4-5-6-7-4-8 (adding number to sum)
- (iii) 1-2-3-4-5-7-4-8 (not adding number to sum)

4) Prepare summary of test cases

Test case Id	Input	Expected o/p	Actual o/p	Result :Pass/Fail
TC1	num_of_entries = -5	0	0	Pass
TC2	num_of_entries = 3 30 60 20	i=1 sum=30 i=2 sum=90 i=3 sum=110	110	Pass
		Sum=110		
TC3	num_of_entries = 1 -30	i=1 sum=0	0	Pass
		Sum=0		

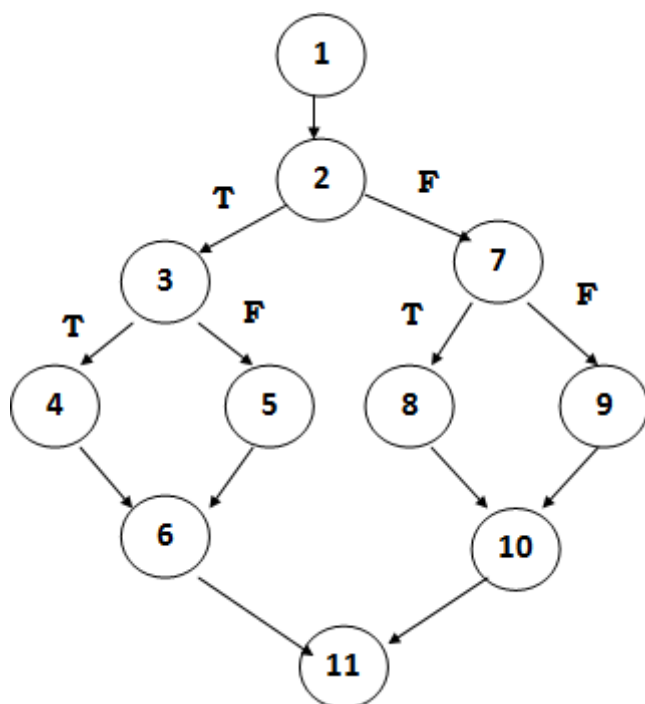
Meaning of cyclomatic complexity value

Complexity	Meaning
1-10	Well written code , testability is high , cost/effort maintain is low
10-20	Moderately complex , testability is medium ,cost/effort to maintain is medium
20-40	Very complex , testability is low ,cost/effort to maintain is High
>40	Not testable ,any amount of money /effort to maintain may not be enough

Problem: Biggest of 3 Numbers

```
1   Read A,B,C
2   If A > B then
3       If A >C then
4           Print " A is greater"
5       Else
6           Print "C is greater"
7   Endif
8   Else
9       If B >C then
10          Print " B is greater"
11         Else
12            Print "C is greater"
13        Endif
14    Endif
```

1. Construct Flow Graph



2. Calculate Cyclomatic Complexity

- | | | |
|-------------------------------------|-----------------|--------------------------|
| 1. $V(G) = E - N + 2$ | $E = 7, N = 11$ | $V(G) = 13 - 11 + 2 = 4$ |
| 2. $V(G) = P + 1$ | $P = 3$ | $V(G) = 3 + 1 = 4$ |
| 3. $V(G) = \text{No of Regions}(R)$ | $R = 4$ | $V(G) = 4$ |

4. Derive Basis Set

- (i) 1-2-3-4-6-11 (A is greater)
- (ii) 1-2-3-5-6-11 (C is greater)
- (iii) 1-2-7-8-10-11 (B is greater)
- (iv) 1-2-7-9-10-11 (C is greater)

5. Summary of Test I/p

Test case Id	Input	Expected o/p	Actual o/p	Result :Pass/Fail
TC1 Path1	A= 12 B=10 C=2	A is greater	A is greater	Pass
TC2 Path2	A= 12 B=10 C= 23	C is greater	C is greater	Pass
TC3 Path3	A= 10 B=12 C= 2	B is greater	B is greater	Pass
TC4 Path4	A= 10 B=12 C= 23	C is greater	C is greater	Pass

Additional White Box Test Design Approaches

- Data Flow and White Box Test Design
- Mutation Testing
- Loop Testing

Test Adequacy Criteria TAC:- (stopping rule)

- Def: Tester need a framework for deciding which structural elements to select as the focus of testing, for choosing the appropriate test data and for deciding when the testing efforts are adequate enough to terminate the process with confidence that the software is working properly.
- It is minimal standards for testing a program
 - Helping testers to select properties of a program to focus on during test
 - Helping testers to select a test data set for a program based on the selected properties
 - Supporting testers with development of quantitative objects for testing
 - Indicating to testers whether or not testing can be stopped for that program

Types of TAC:

1. Program Based TAC : focus on structural properties of program , includes logic ,control structure , data flow
2. Specification based TAC: focus on program specification
3. Random TAC: ignores both program structure& specification.
Ex: TAC focus on statement/branch properties

“A test data set is statement, or branch , adequate if a test set T for program P Causes all the statements, or branches to be executed respectively”

Coverage Analysis: The TAC & the requirement that certain features of the code are to be exercised by the test case, also named as coverage criteria

Degree of coverage : when a coverage related testing goal is expressed as a percent.

degree of coverage < 100% due to the following:-

1. The Nature of the Unit
 - i. Some statements/branches may not be reachable
 - ii. The unit may be simple, and not mission or safety , critical and so complete coverage is thought to be unnecessary
2. The lack of resources
 - i. The time set aside for testing is not adequate to achieve 100% coverage
 - ii. There are not enough trained testers to achieve complete coverage for all the units
 - iii. There is a lack of tools to support complete coverage.
3. Other project related issued such as timing, scheduling and marketing constraints.

Ex: 4 branches in s/w unit

2 are executed by planned set of test cases Degree of branch coverage : 50%	Coverage goal is not met
Develop Additional test cases & re execute the test cases	Continue until desired degree is obtained

Evaluating Test Adequacy Criteria :TAC hierarchy

- Tester can select appropriate criterion using the hierarchy
- Criteria at the top includes the Criteria at the Bottom , for example All def-use path adequacy means - tester achieved branch & statement adequacy
- Each Adequacy Criteria has both strength and weakness
- Stronger criteria → tester need more time and resource

Example : (Sample code with data flow information)

def → variable defined

use → variable Used

p-use → Predicate Use , variable used in condition

c-use → Computation use , variable used in calculation

1 sum=0	sum, def
2 read (n)	n, def
3 i=1	i, def
4 while (i <=n)	i, n p-use
5 read (number)	number, def
6. sum=sum+number	sum, def, sum, number, c-use
7 i=i+1	i, def, c-use
8 end while	
9 print (sum)	sum, c-use

Ex: DU Chain (Def-Use Path) Chain in Data flow Testing

Def-Use Path a path from a variable definition to a use is called a def-use path

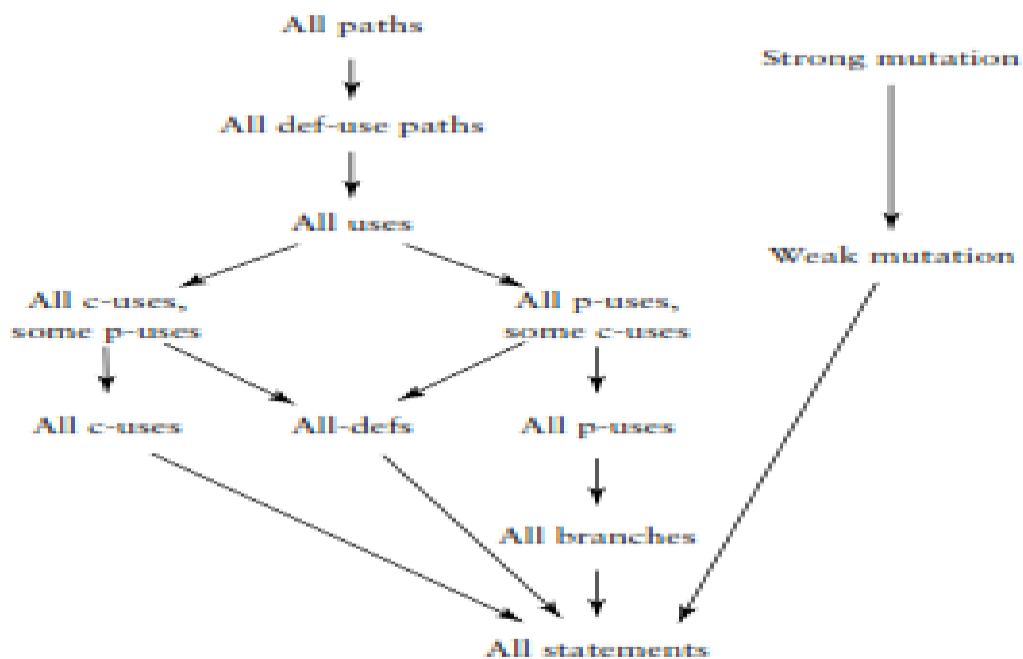
Table for n		
pair id	def	use
1	2	4

Table for number		
pair id	def	use
1	5	6

Table for sum		
pair id	def	use
1	1	6
2	1	9
3	6	6
4	6	9

Table for i		
pair id	def	use
1	3	4
2	3	7
3	7	7
4	7	4

Partial Ordering for Test Adequacy Criteria



Axioms

set of axioms that allow testers to formalize properties which should be satisfied by any good program-based test data adequacy criterion

Testers can use the axioms to

- recognize both strong and weak adequacy criteria;

- focus attention on the properties that an effective test data adequacy criterion should exhibit;
- select an appropriate criterion for the item under test;
- stimulate thought for the development of new criteria;

The axioms are based on the following set of **assumptions**

- (i) programs are written in a structured programming language;
- (ii) programs are SESE (single entry/single exit);
- (iii) all input statements appear at the beginning of the program;
- (iv) all output statements appear at the end of the program.

The **axioms/properties** described by Weyuker are the following

1. Applicability Property
2. Non exhaustive Applicability Property
3. Monotonicity Property
4. Inadequate Empty Set
5. Anti extensionality Property
6. General Multiple Change Property
7. Anti decomposition Property
8. Anti composition Property
9. Renaming Property
10. Complexity Property
11. Statement Coverage Property

Sample test data adequacy criteria and axiom satisfaction

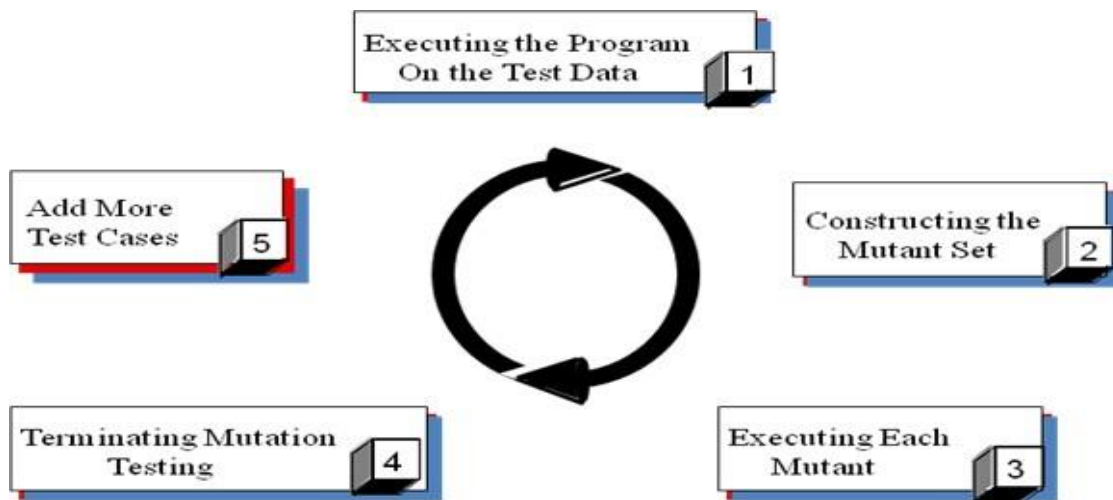
	Statement	Branch	Mutation
Axiom 1	No	No	Yes
Axiom 2	Yes	Yes	Yes
Axiom 3	Yes	Yes	Yes
Axiom 4	Yes	Yes	Yes
Axiom 5	Yes	Yes	Yes
Axiom 6	Yes	Yes	Yes
Axiom 7	No	No	Yes
Axiom 8	No	No	Yes

Mutation Testing

- is a testing technique that focuses on measuring the adequacy of test cases.
- A test case is *adequate* if it is useful in detecting faults in a program.
- A test case can be shown to be adequate by finding at least one mutant program that generates a different output than does the original program for that test case.

- If the original program and all mutant programs generate the same output, the test case is *inadequate*.

Basic Steps



Kinds of Mutation

A mutation is a small change in a program.

Value Mutations: these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds { being one out on the start or finish is a very common error.

Decision Mutations: this involves modifying conditions to reflect potential slips and errors in the coding of conditions in programs, e.g. a typical mutation might be replacing a > by a < in a comparison.

Statement Mutations: these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Example of Testing By Decision Mutation

First test data set--M=1, N=2 , the original function returns 2

- five mutants: replace ">" operator in if statements by (>,<,<=or=)

Program	Mutants		
	Mutants	Outputs	Comparison
function MAX(M ,N:INTEGER)	if M>=N then	2	live
return INTEGER is	if M<N then	1	dead
begin	if M<=N then	1	dead
if M>N then	if M=N then	2	live
return M;	if M< >N then	1	dead
else			
return N;			
end if;			
end MAX;			

- Executing each mutant: adding test data M=2, N=1 will eliminate the latter live mutant, but the former live mutant remains live because it is equivalent to the original function. No test data can eliminate it.