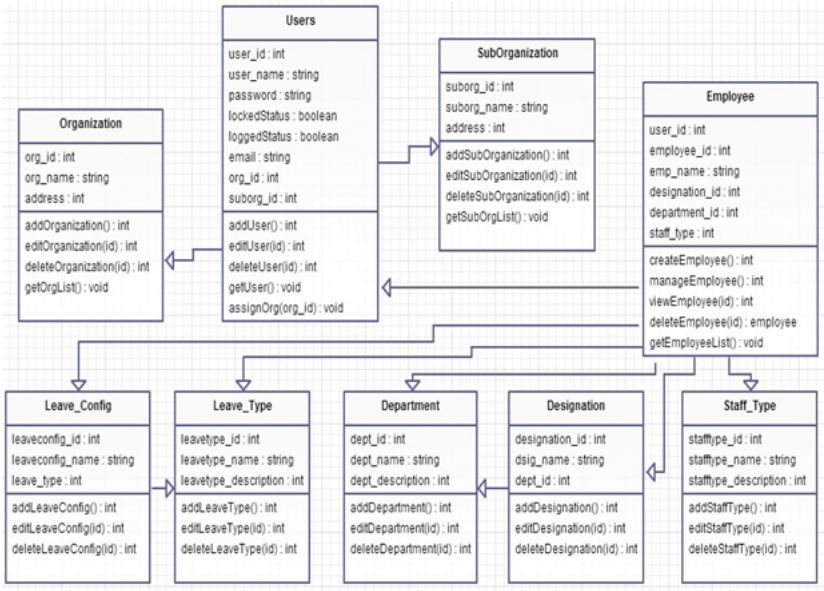
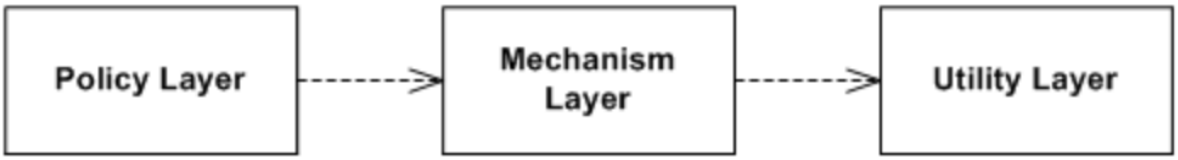
Coupling: Dependency of an entity over other entities.
Class Coupling: A class depends on other classes, for example, uses functions, takes arguments, etc
Module Coupling: In Java class of one package is coupled with class of other package.

Instability value = Efferent (out-going)/ (Efferent+ Afferent (in-coming)) = Ce / (Ce + Ca)
Values in between 0 & 1.
0 means absolutely stable class (it depends on nobody, everybody depends on it)
1 means absolutely instable class (it depends on other classes, whenever other classes are changed, it has to be changed. If this class is changed, others aren't affected)



Example: Instability Value of Class: Employee = 6 / (6+0) = 1 (instable)
Instability Value of Class: Users = 2 / (2+1) = 0.66667
Instability Value of Class: Leave_type = 0 / (0+2) = 0 (stable)

Traditional Dependency:

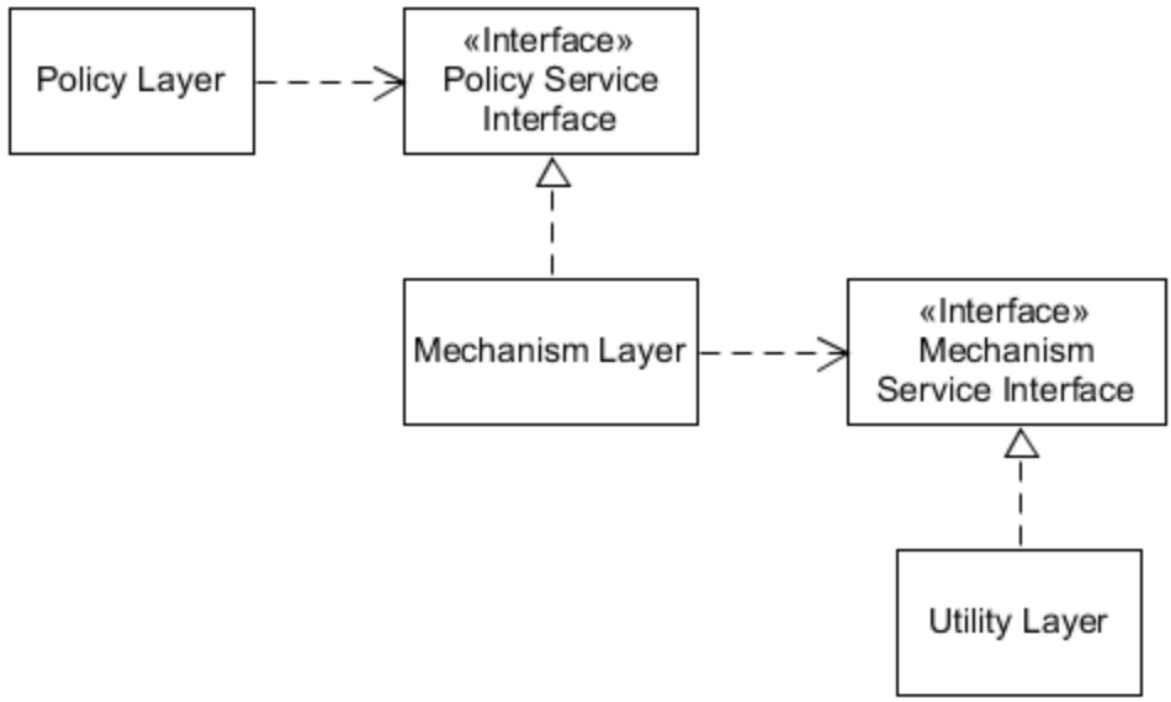


Policy Layer (Instability) = 1 / (1+0) = 1 (instable)
Mechanism Layer = 1 / (1+1) = 0.5
Utility Layer = 0 / (0 + 1) = 0 (stable)

Dependency inversion pattern (DIP)

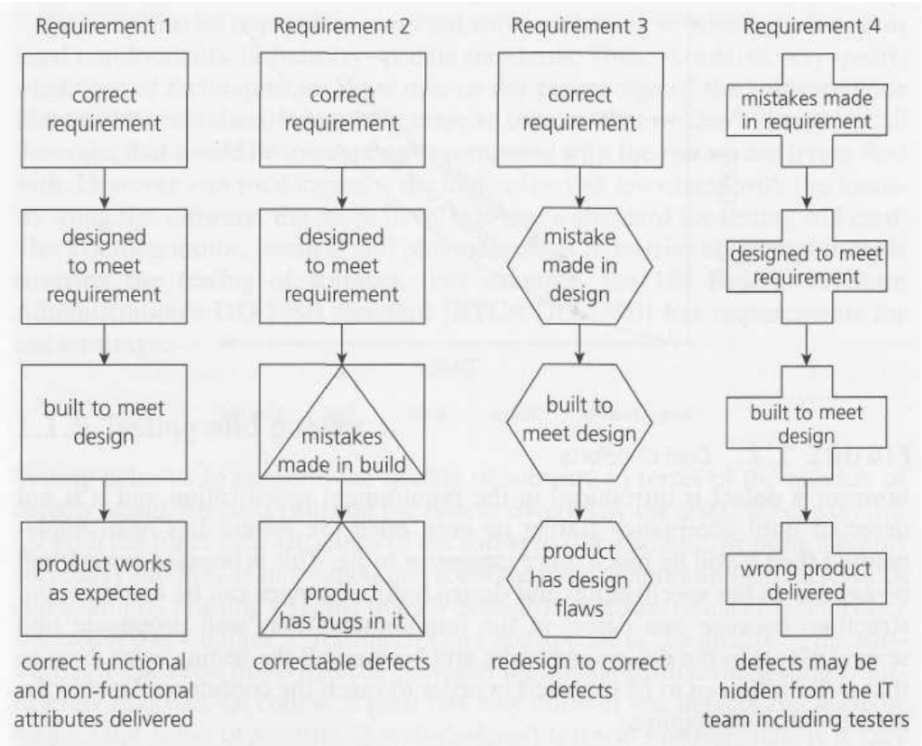
DIP allows to introduce abstract layers to change the traditional dependencies onto abstract layers.

DIP approach:

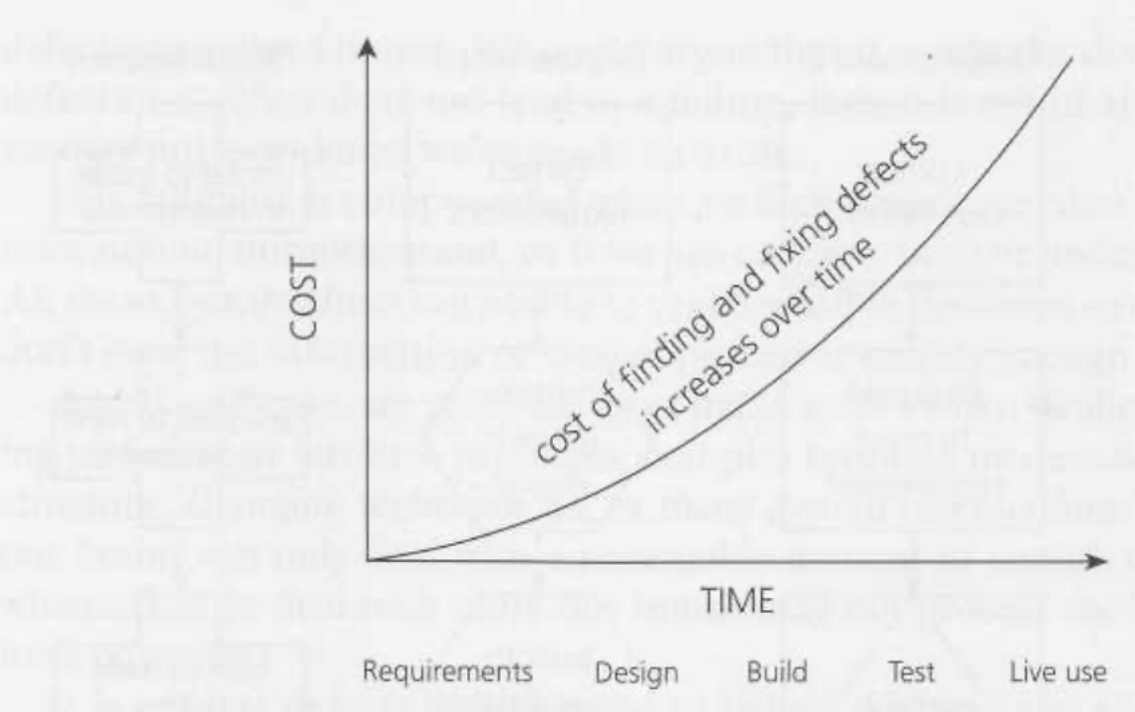


Policy Layer (Instability) = 0 / (0+1) = 0 (stable)
Mechanism Layer = 1 / (1+1) = 0.5
Utility Layer = 1 / (1 + 0) = 1 (instable)

Errors:



Cost of defects at different SDLC lifecycle


COST
cost of finding and fixing defects
increases over time
TIME
Requirements
Design
Build
Test
Live use

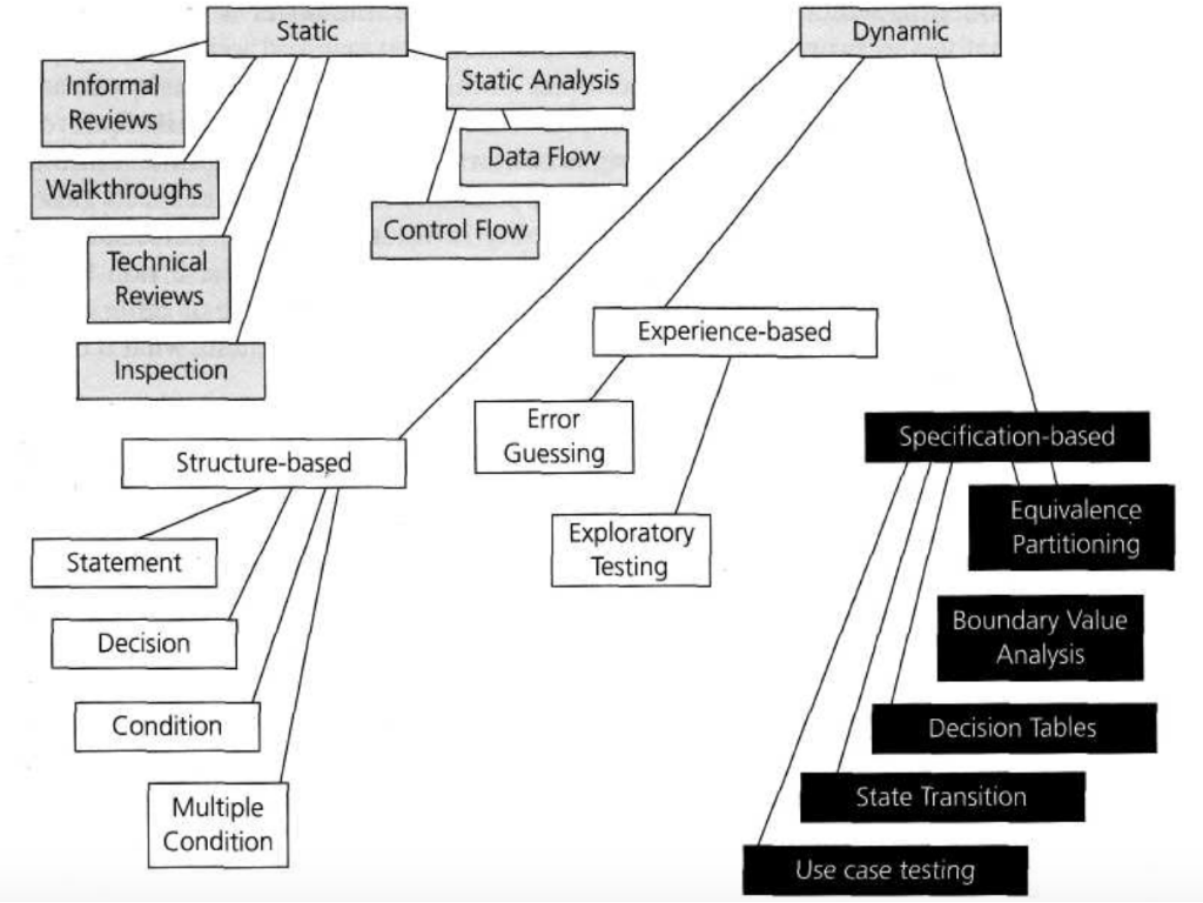
Testing Principles

Principle 1:	**Testing shows presence of defects**	Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness.
Principle 2:	**Exhaustive testing is impossible**	Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, we use risks and priorities to focus testing efforts.
Principle 3:	**Early testing**	Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.
Principle 4:	**Defect clustering**	A small number of modules contain most of the defects discovered during pre-release testing or show the most operational failures.
Principle 5:	**Pesticide paradox**	If the same tests are repeated over and over again, eventually the same set of test cases will no longer find any new bugs. To overcome this 'pesticide paradox', the test cases need to be regularly reviewed and revised, and new and different tests need to be written to exercise different parts of the software or system to potentially find more defects.
Principle 6:	**Testing is context dependent**	Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce site.
Principle 7:	**Absence-of-errors fallacy**	Finding and fixing defects does not help if the system built is unusable and does not fulfill the users' needs and expectations.

Discussion:

- Why Testing is important/essential?
- What is cost of the defects?
- What is relationship between testing and quality?
- What are testing objectives?
- What is 'root cause analysis' ? What's its importance in testing?
- How much testing is enough?

Testing Techniques



Static Analysis:
Control Flow: Cyclomatic Complexity - The **cyclomatic complexity** metric is based on the number of decisions in a program.

$$M = E - N + 2P,$$

where

E = the number of edges of the graph.

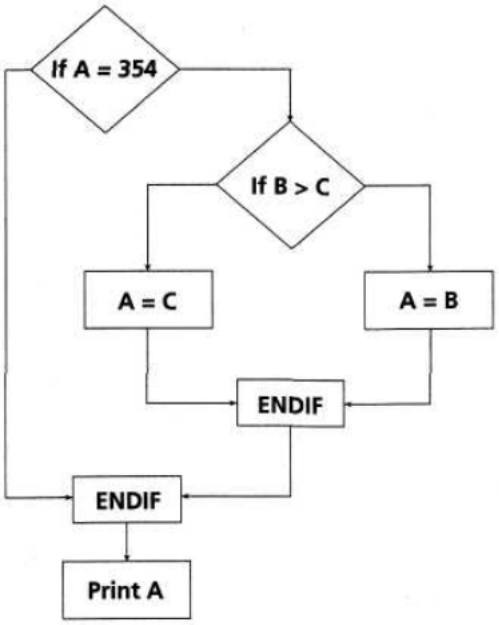
N = the number of nodes of the graph.

P = the number of connected components.

Another example:
Consider the program

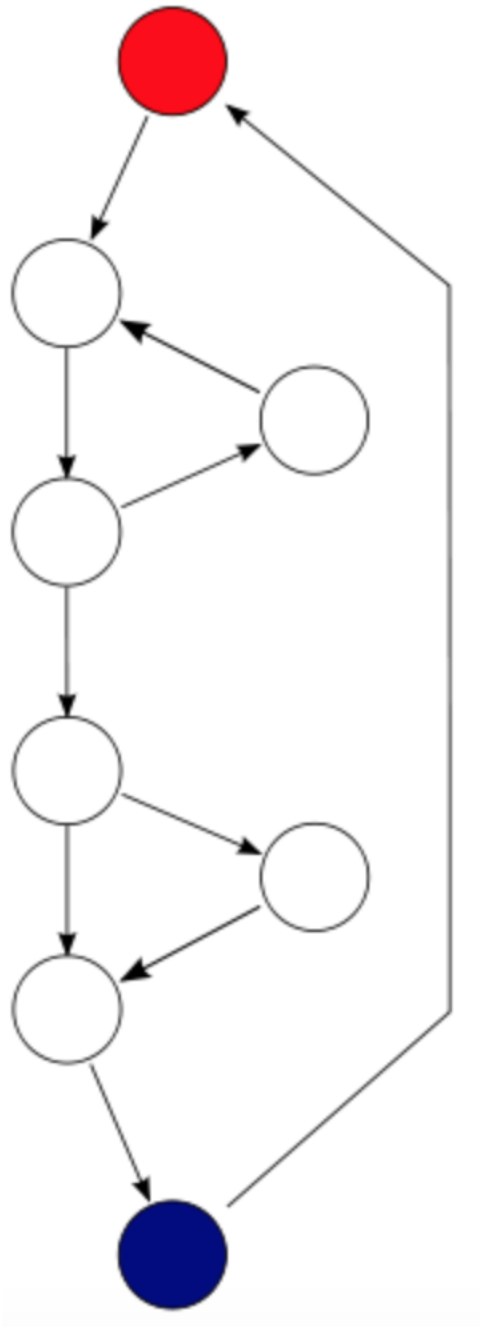
```
IF A = 354
 THEN IF B > C
  THEN A = B
  ELSE A = C
 ENDIF
ENDIF
Print A
```

Its graph diagram



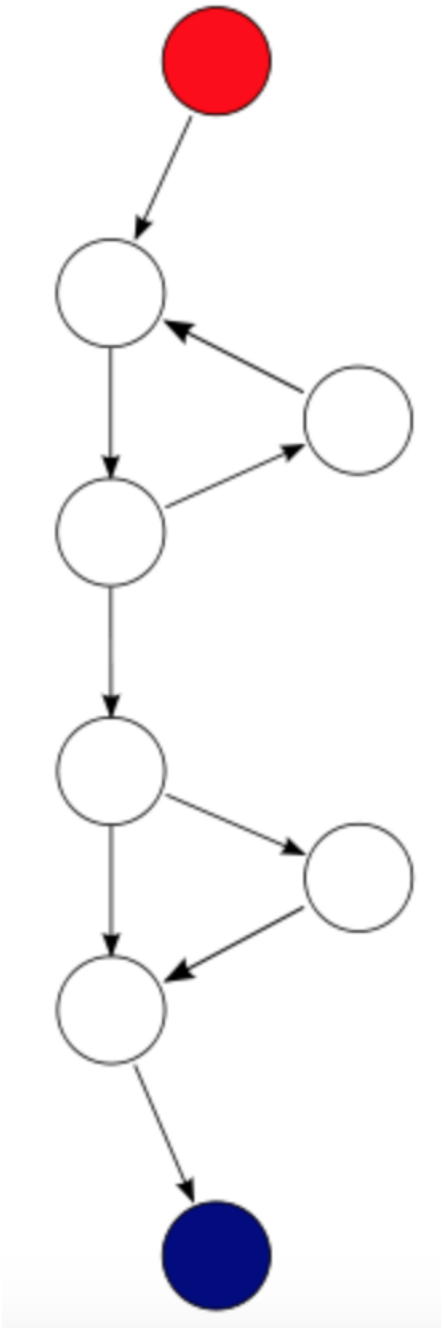
Cyclomatic Complexity Matric = ???

Function has 1 complexity, add 1 when there is condition and add 1 when there is loop.
Example:



Cyclomatic Complexity = 4

Calculate Cyclomatic Complexity



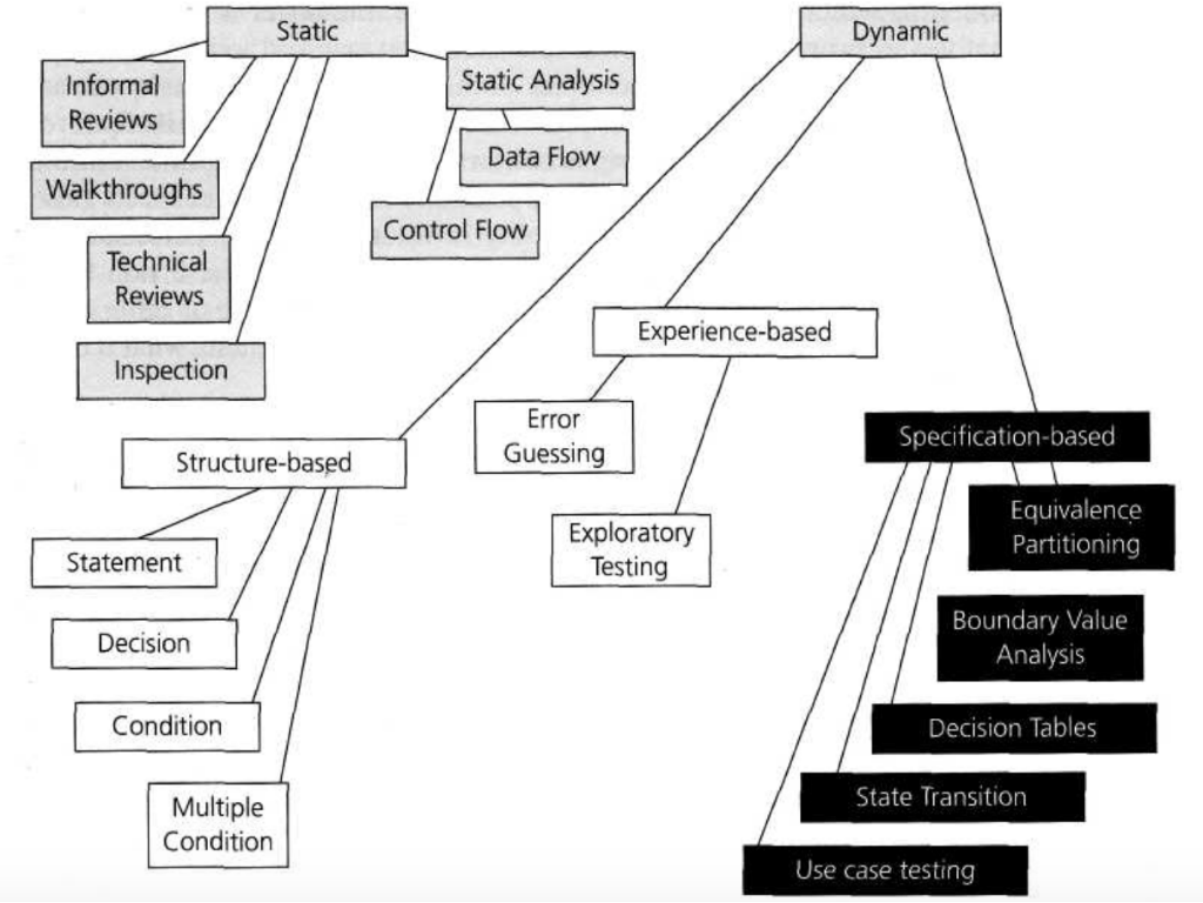
Data Flow: Flow of data (flow chart for data members)
Data Structure: which data structure has been used.

Dynamic Testing Techniques: code is being executed.
Structure-based testing (white box or glass box): assessment of how software is implemented. Internal structure of the software.
Test coverage: it's measure that indicates the coverage of the items in the structure of the software.
Coverage = (number of coverage items exercised/ total coverage items) * 100%

Specification-based testing (black box): functional requirement assessment. Deals with inputs/outputs. It has nothing to do with how it is implemented. It only assesses what it does.

Experience-based testing: it requires experience, domain knowledge as well as business knowledge to derive test cases.

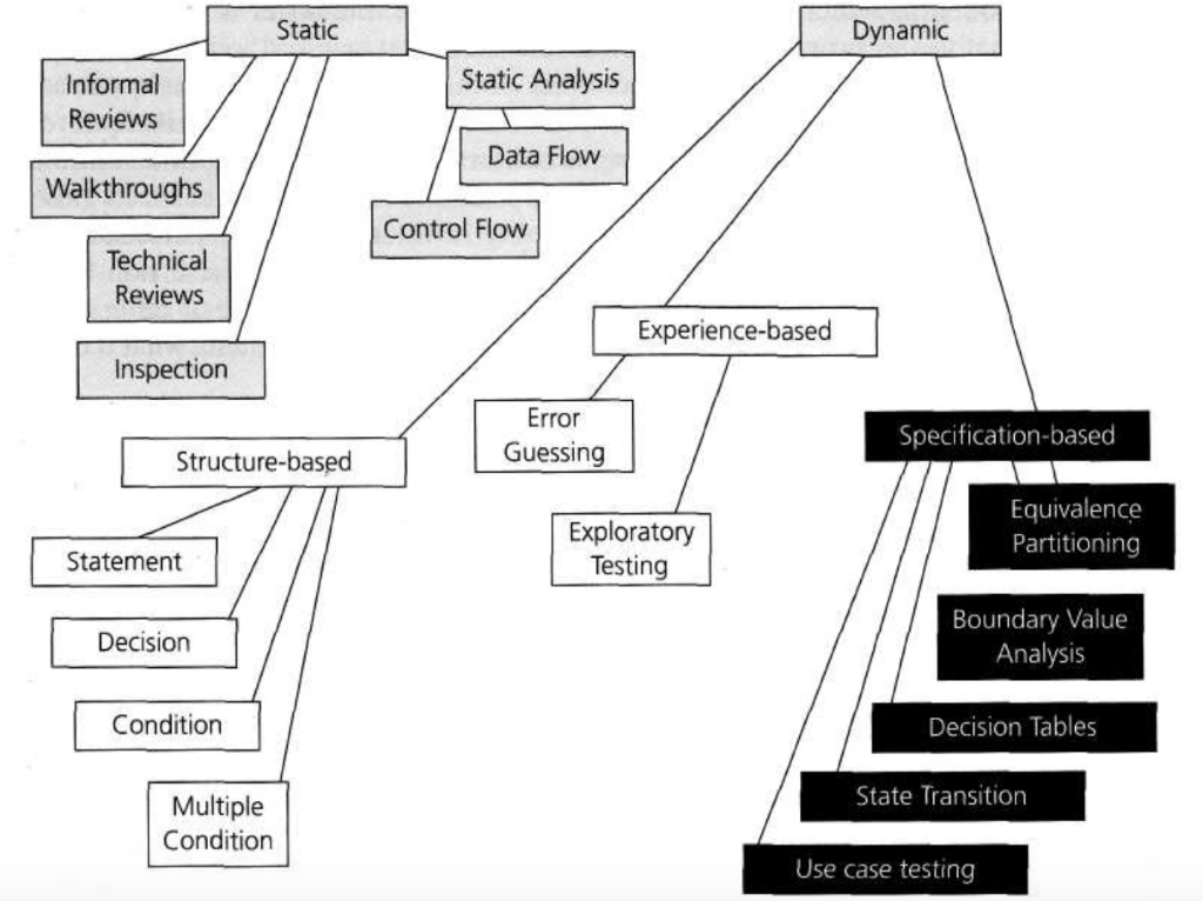
Static Testing Techniques:



Static Techniques are non-execution techniques.
A walkthrough is characterized by the author of the document under review guiding the participants through the document and his or her thought processes, to achieve a common understanding and to gather feedback. A walkthrough is especially useful for higher-level documents, such as requirement specifications and architectural documents.

A **technical review** is a discussion meeting that focuses on achieving consensus about the technical content of a document.
Inspection is the most formal review type. The document under inspection is prepared and checked thoroughly by the reviewers before the meeting, comparing the work product with its sources and other referenced documents, and using rules and checklists.
Success/Goal of Software Engineer?

Dynamic Testing Techniques



Dynamic Testing Techniques: code is being executed.

Specification-based testing (black box or Behavioral Testing): functional requirement assessment. Deals with inputs/outputs. It has nothing to do with how it is implemented. It only assesses what it does.

- equivalence partitioning (EP);
- boundary value analysis (BVA);
- decision tables (DT);
- state transition testing (STT).

Equivalence Partitioning (EP)

EP divides the software into smaller classes/partitions to test, thus, its equivalence partitions or equivalence classes.

For example, A banking software calculates interest rate of 3% when balance is in between (0-100)USD, 5% when balance in between (100-1000)USD, and 7% for above 1000USD.

EP allows to have four partitions/classes such as:

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
-$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

Assumptions made in EP partitions:

- Decimal points (two digits after decimal point)
- All valid numeric inputs (ignoring character say 'a' as input)
- Maximum value is ignored (undefined maximum balance value)

Note:- Mention the partitions so that tester know the values in order to test each partition.

Boundary Value Analysis (BVA)

The analysis at the range values to test in software/program.
For example, a printer software takes valid range of values upto two digits in between 1-99 (inclusive).

Invalid	Valid	Invalid
0 | 1	99 | 100	

Baking Software running example of EP:

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
–$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

The following table shows the EP and BVA for banking software (running example).

Test conditions	Valid partitions	Invalid partitions	Valid boundaries	Invalid boundaries
Balance in aUoUnt	$0.00 $100.00	< $0.00	$0.00	-$0.01
	$100.01-$999.99	>$Max	$100.00	$Max+0.01
	$1000.00- $Max	non-integer (if balance is an input field)	$100.01	
			$999.99	
			$1000.00	
			$Max	
Interest rates	3%	Any other value	Not applicable	Not applicable
	5%	Non-integer		
	7%	No interest calculated		

These conditions helps in designing test cases.
Exercise: Design 2-3 test cases for this banking software in which a field asks for balance value and provides the interest rate.

Decision Tables (DT) - Also known as 'Cause-effect' table

This technique is good for combination of things (I.e., inputs) to deal business logic or business rules using 2^2 formula. For example, 2 input combination results into 4 combinations and 3 input results into 8 combinations, likewise 4 inputs results into 16 combinations.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount		Y		
Process term			Y	
Error message	Y			Y

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Result	Error message	Process loan amount	Process term	Error message

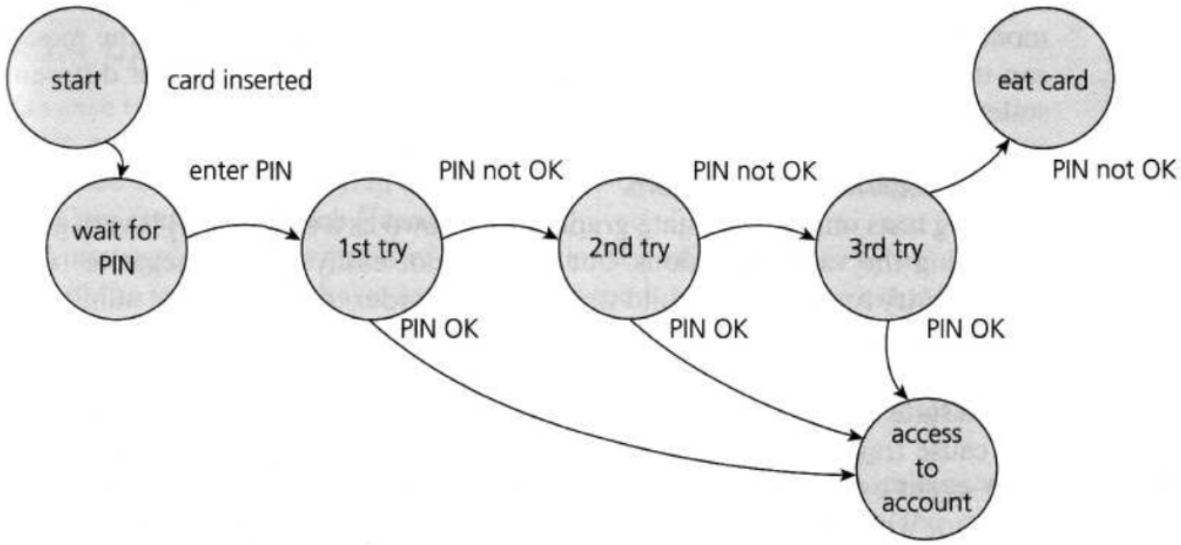
Another example of credit card

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

State Transition Testing (STT)

State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'.

Any system where a different output is obtained for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a **state diagram.**



The **state table** lists all the states down one side of the table and all the events that cause transitions along the top (or vice versa). Each cell then represents a state-event pair.

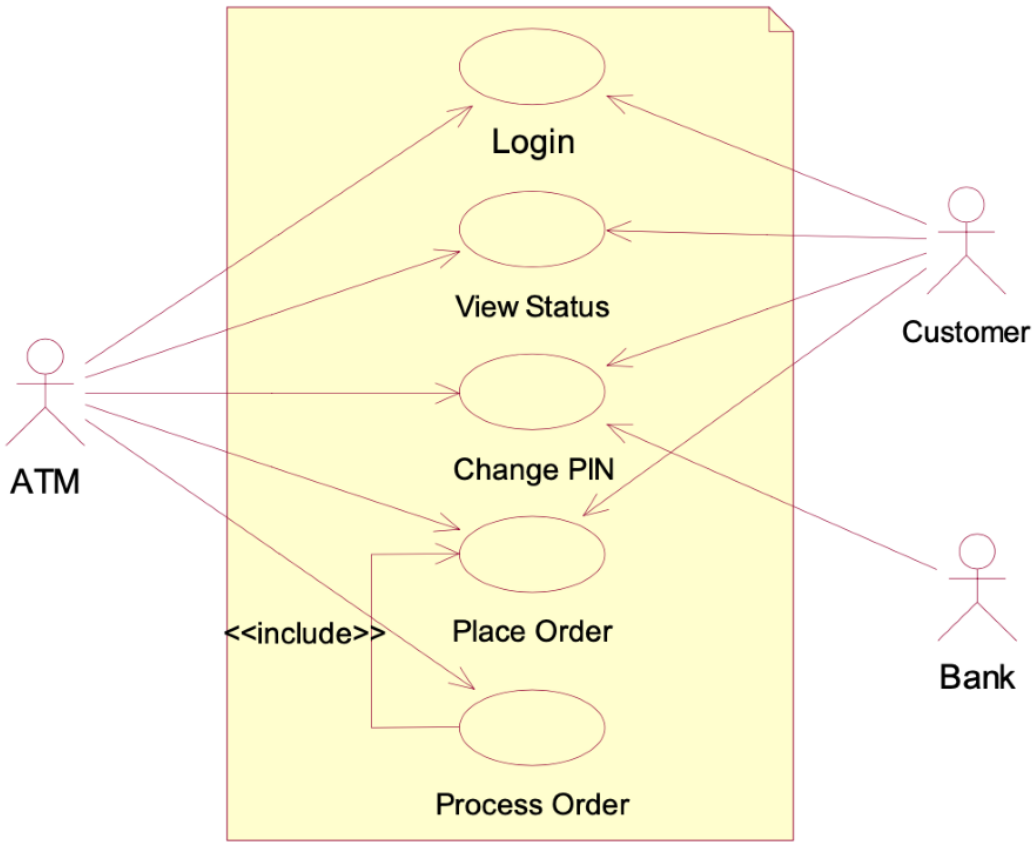
	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	–	–
S2) Wait for PIN	–	S6	S3
S3) 1st try invalid	–	S6	S4
S4) 2nd try invalid	–	S6	S5
S5) 3rd try invalid	–	–	S7
S6) Access account	–	?	?
S7) Eat card	S1 (for new card)	–	–

Use case Testing

Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish.

A use case is a description of a particular use of the system by an actor (a user of the system). Each use case describes the interactions the actor has with the system in order to achieve a specific task (or, at least, produce something of value to the user). Actors are generally people but they may also be other systems. Use cases are a sequence of steps that describe the interactions between the actor and the system.

Use Case Diagram -

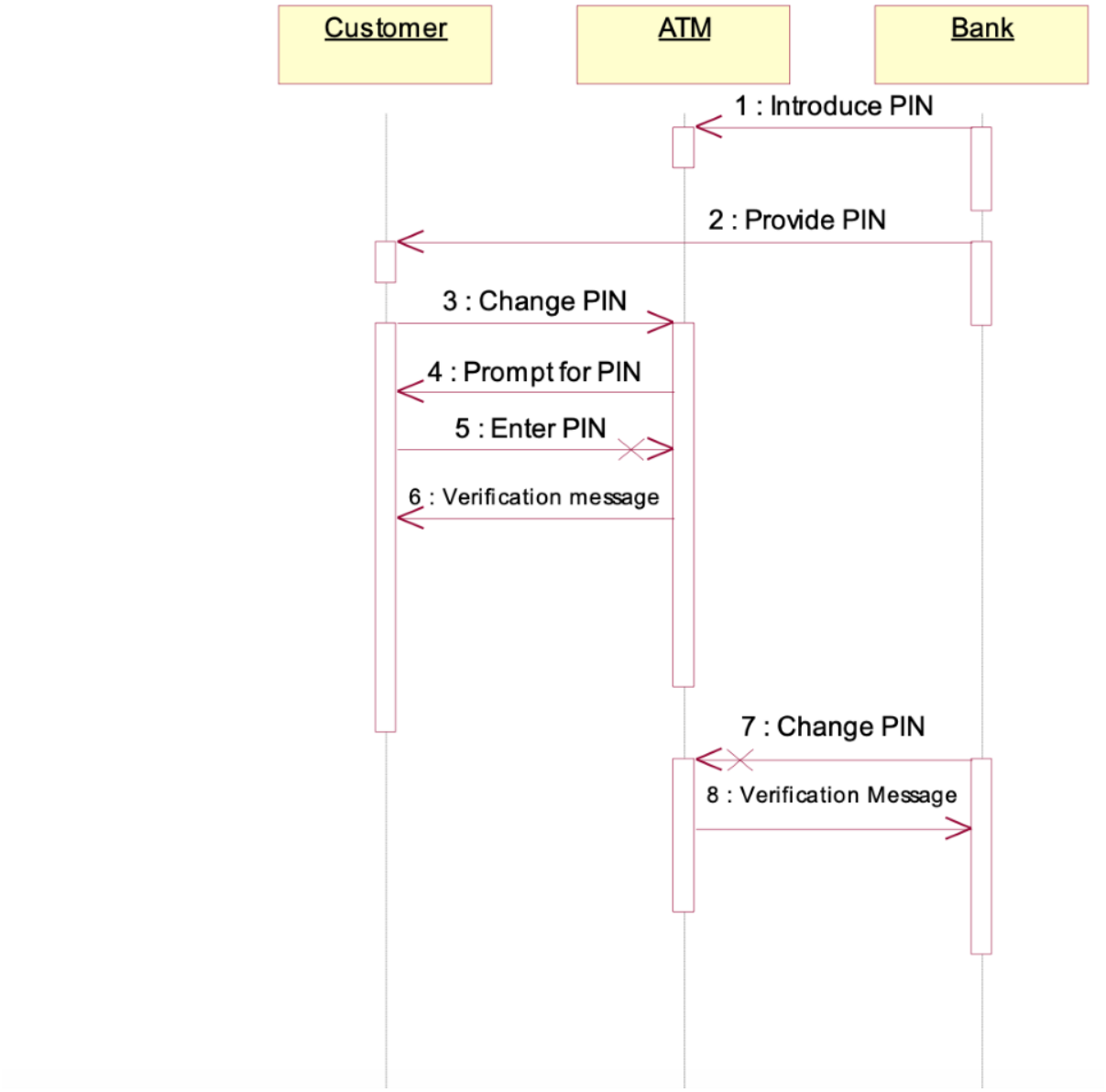


Use Case 3 – Change PIN
Actors – Customer (Initiator), ATM, BANK
Purpose – Change the current PIN code
Overview – Customer selects operation of changing the PIN code / Bank (Officer) introduce the PIN code or change it

General Course of Action –

Actor's Action	**System Response**
Customer selects the operation of changing PIN code	System prompt to enter new PIN code
[Customer enters new PIN]	System assigns new PIN to the account and messages customer as success
Bank (Officer) introduces or changes current PIN code on request of customer	System assigns new PIN to the customer account and messages as success

Sequence Diagram of Change PIN –



Structure-based testing (white box or glass box): assessment of how software is implemented. Internal structure of the software. It helps to write additional test cases and test the project/software in order to increase test coverage possible 100% coverage.

Test coverage: it's measure that indicates the coverage of the items in the structure of the software.
Coverage = (number of coverage items exercised/ total coverage items) * 100%

Note:- 100% coverage does *not* mean 100% tested!

Specification based testing can be measured in terms of coverage.

- EP: % of tested partitions
- BVA: % of boundary tests tested

- DT: % of business rules tested
- State Transition Testing: % of states tested

Coverage from user/customer point of view is the % functionality tested (I.e., fitness for purpose/requirements/specifications)

However, structural testing focus on **code coverage** such as statements, decisions, conditions, loops. It is often done of coverage measurement tools.

Statement Coverage

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

For example, Pseudo Code:

Read A
Read B
IF A>B Then C=0
ENDIF

Statement Coverage: To get 100% coverage of this Pseudo code, A must be greater than B. Say A=12, B=10. (Note:- It's structural design test, thus, introduce values to get maximum code coverage)

Another Example,

```
1 READ A
2 READ B
3 C=A+2*B
4 IF C> 50 THEN
5       PRINT large C
6 ENDIF
```

Test Sets:
Test 1.1: A = 2, B = 3
Test 1.2: A = 0, B = 25
Test 1.3: A = 47, B = 1

- In Test 1.1, the value of C will be 8, so it will cover the statements on lines 1 to 4 and line 6.
- In Test 1.2, the value of C will be 50, so it will cover exactly the same statements as Test 1.1.
- In Test 1.3, the value of C will be 49, so again it will cover the same statements.

Test 1.4: A = 20, B = 25
This time the value of C is 70, so it will print 'Large C and it will have exercised all six of the statements, so now statement coverage = 100%.

Decision Coverage

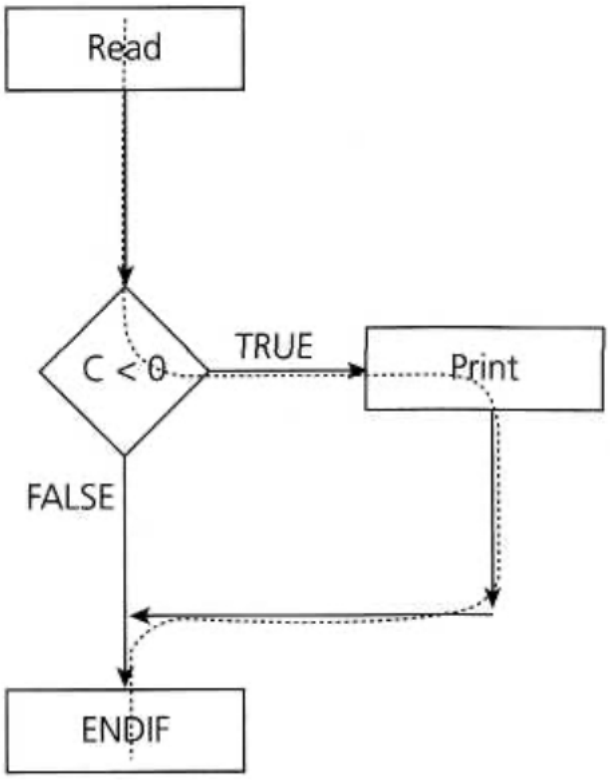
$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

Decision coverage is stronger than statement coverage. Decision coverage measures the coverage of conditional branches.
100% decision coverage always guarantees 100% statement coverage.

Example,

```
1  READ A
2  READ B
3 C=A-2*B
4 IFC <0THEN
5 PRINT "C negative"
6 ENDIF
```

TEST SET
Test 2.1: A = 20, B = 15 (TRUE Decision Flow Diagram)
Test 2.2: A = 10, B = 2 (False Decision Flow Diagram)



Other Structure based techniques
Other control-flow code-coverage measures include linear code sequence and jump (LCSAJ) coverage, condition coverage, multiple condition coverage (also known as condition combination coverage) and condition determination coverage (also known as multiple condition decision coverage or modified con- dition decision coverage, MCDC). This technique requires the coverage of all conditions that can affect or determine the decision outcome.

Note:- Sometimes any structure-based technique is called 'path testing'

Experience-based testing: it requires experience, domain knowledge as well as business knowledge to derive test cases.
The reason is that some defects are hard to find using more system- atic approaches, so a good 'bug hunter' can be very creative at finding those elusive defects.

Error guessing is a technique that should always be used as a complement to other more formal techniques. The success of error guessing is very much dependent on the skill of the tester, as good testers know where the defects are most likely to be present.
There are no rules for error guessing. The tester is encouraged to think of situations in which the software may not be able to cope. Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data (e.g. alphabetic characters where numeric are required)

Exploratory testing is a hands-on approach in which testers are involved in minimum planning and maximum test execution. The planning involves the cre- ation of a test charter, a short declaration of the scope of a short (1 to 2 hour) time-boxed test effort, the objectives and possible approaches to be used.

The test design and test execution activities are performed in parallel typi- cally without formally documenting the test conditions, test cases or test scripts.

This is an approach that is most useful when there are no or poor specifica- tions and when time is severely limited.

Which technique is best?

TEST TYPES: THE TARGETS OF TESTING

A **test type** is focused on a particular test objective

1. **Functional**
2. **non-functional**
3. **Structural**
4. **change-related**

1. **Testing of Function (functional testing)**

The function of a system (or component) is 'what it does'.
Functional testing considers the specified behavior and is often also referred to as **black-box testing.** This is not entirely true, since black-box testing also includes non-functional testing.
Function (or functionality) **testing** can, based upon ISO 9126, be done focus- ing on suitability, **interoperability, security,** accuracy and compliance. Security testing, for example, investigates the functions (e.g. a firewall) relating to detection of threats, such as viruses, from malicious outsiders.

Testing functionality can be done from two perspectives: requirements-based or business-process-based.
Requirements-based testing uses a specification of the functional require- ments for the system as the basis for designing tests. A good way to start is to use the table of contents of

the requirements specification as an initial test inventory or list of items to test (or not to test).

Business-process-based testing uses knowledge of the business processes. Business processes describe the scenarios involved in the day-to-day business use of the system. For example, a personnel and payroll system may have a busi- ness process along the lines of: someone joins the company, he or she is paid on a regular basis, and he or she finally leaves the company.

1.1. **Non-Functional testing as functional testing**
Non-functional testing, as functional testing, is performed at all test levels. Non-functional testing includes, but is not limited to, **performance testing, load testing, stress testing,** usability testing, maintainability testing, reliability testing and portability testing. It is the testing of 'how well' the system works.

The ISO 9126 standard defines six quality characteristics

- **functionality**, which consists of five sub-characteristics: suitability, accuracy, security, interoperability and compliance; this characteristic deals with functional testing;
- **reliability,** which is defined further into the sub-characteristics maturity (robustness), fault-tolerance, recoverability and compliance;
- **usability,** which is divided into the sub-characteristics understandability, learnability, operability, attractiveness and compliance;
- **efficiency,** which is divided into time behavior (performance), resource uti lization and compliance;
- **maintainability,** which consists of five sub-characteristics: analyzability, changeability, stability, testability and compliance;
- **portability,** which also consists of five sub-characteristics: adaptability, installability, co-existence, replaceability and compliance.

Fundamental Testing Process:
- planning and control;
- analysis and design;
- implementation and execution;
- evaluating exit criteria and reporting;
- test closure activities.