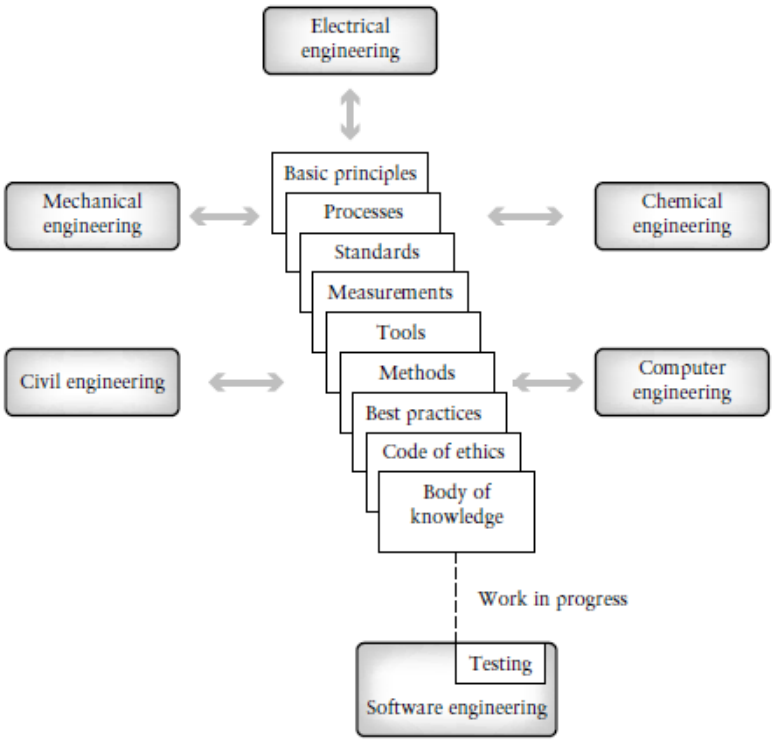
Unit-I Introduction

1.1 The Evolving Profession of Software Engineering

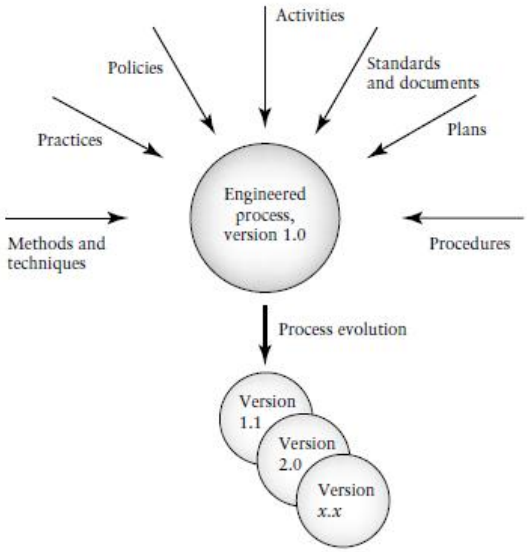
- the development process is well understood;
- projects are planned;
- life cycle models are defined and adhered to;
- standards are in place for product and process;
- measurements are employed to evaluate product and process quality;
- components are reused;



Elements of the engineering disciplines

The Role of Process in Software Quality

Process, in the software engineering domain, is the set of methods, practices, standards, documents, activities, policies, and procedures that software engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents, code, and manuals.

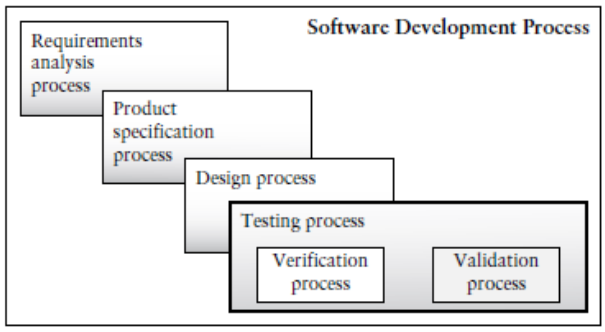


Components of an engineering process

Testing as a Process

Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements.

Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.



Example process embedded in the software development

Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software.

Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

An Overview of the Testing Maturity Model

1. **T**here is a demand for software of high quality with low defects;
2. **P**rocess is important in the software engineering discipline;
3. **S**oftware testing is an important software development sub process;
4. **E**xisting software evaluation and improvement models have not adequately addressed testing issues.

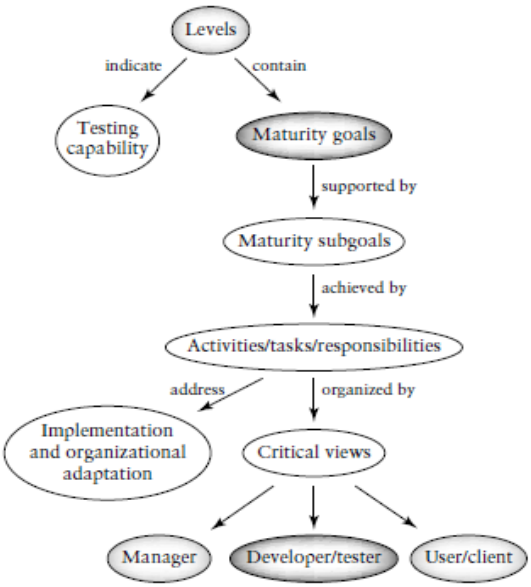
In the testing domain possible benefits of test process improvement are the following:

- smarter testers
- higher quality software
- the ability to meet budget and scheduling goals
- improved planning
- the ability to meet quantifiable testing goals

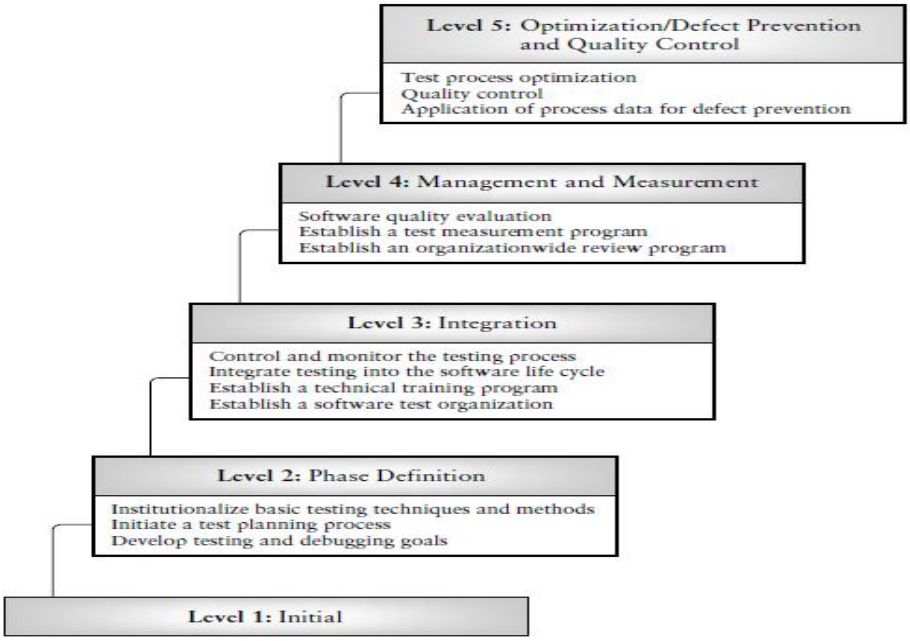
TMM L e v e l s

A set of maturity goals. The maturity goals identify testing improvement goals that must be addressed in order to achieve maturity at that level. To be placed at a level, an organization must satisfy the maturity goals at that level. The TMM levels and associated maturity goals are shown in Figure.

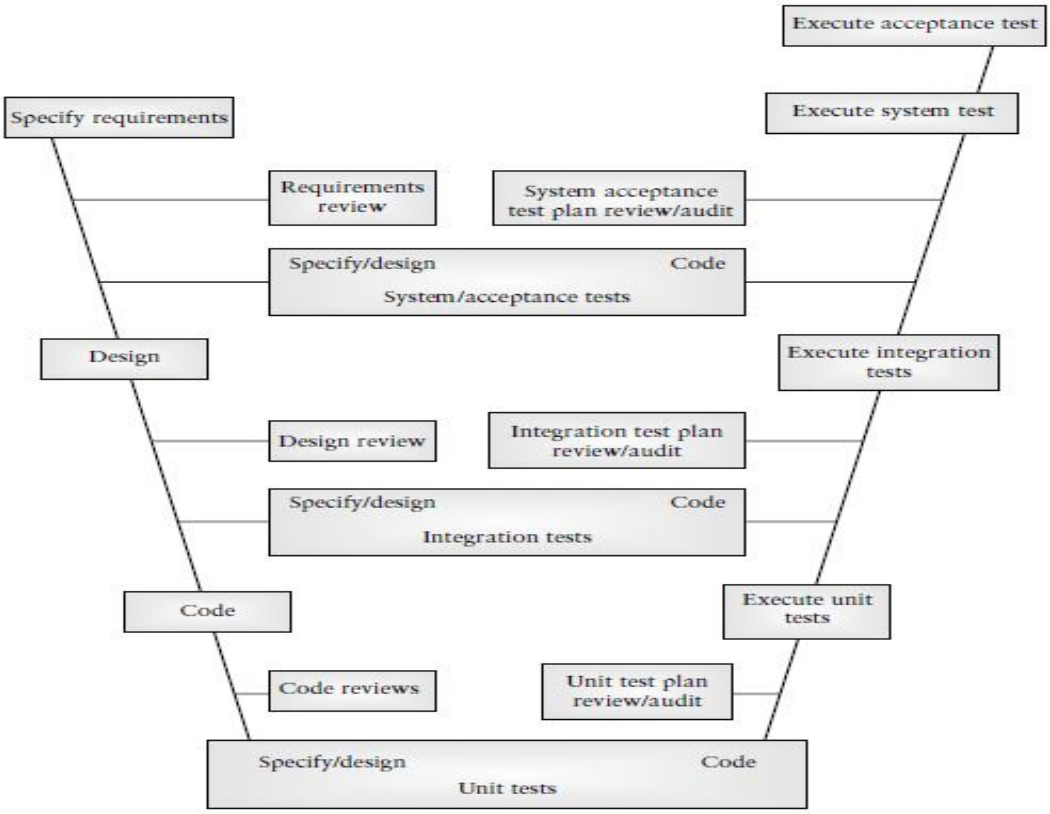
- *Supporting maturity subgoals.* They define the scope, boundaries and needed accomplishments for a particular level.
- *Activities, tasks and responsibilities (ATR).* The ATRs address implementation and organizational adaptation issues at each TMM



The internal structure of TMM maturity levels



The s-level structure of the testing maturity model



The Extended/Modified V-model

1.2 Testing Fundamentals

Error & Faults (Defects)

- An error is a mistake, misconception, or misunderstanding on the part of a software developer.
- A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

Failures

- A failure is the inability of a software system or component to perform its required functions within specified performance requirements

Test Cases

- A test case in a practical sense is a test-related item which contains the following information:
- *A set of test inputs.* These are data items received from an external source by the code under test. The external source can be hardware, software, or human.
- *Execution conditions.* These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
- *Expected outputs.* These are the specified results to be produced by the code under test.

Test

- A test is a group of related test cases, or a group of related test cases and test procedures

Test Oracle

- A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

Test Bed

- A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.

Software Quality

- Quality relates to the degree to which a system, system component, or process meets specified requirements.
- Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.
- A metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute

- A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute

Software Quality Assurance Group

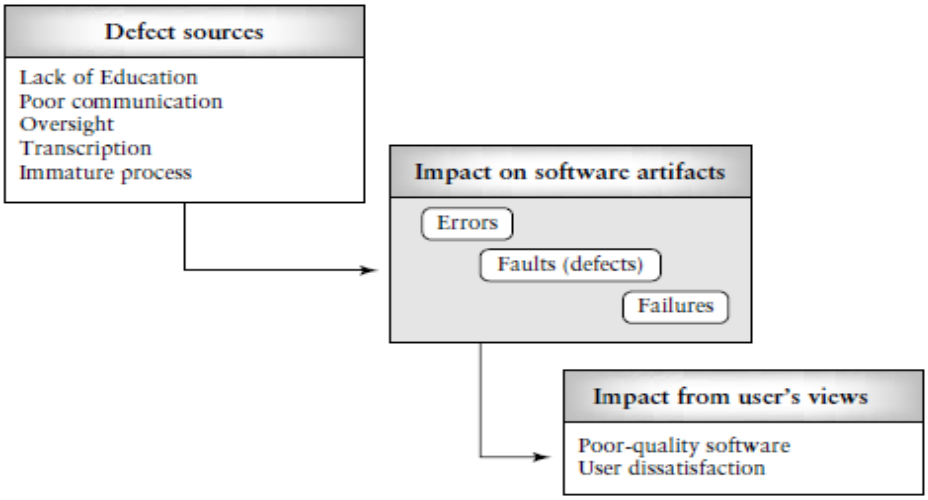
- The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so hat the resulting software conforms to established technical requirements.

Reviews

- A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

1.3 D E F E C T S , H Y P O T H E S E S ,A N D T E S T S

Origins of Defects



Hypotheses

- design test cases;
- design test procedures;
- assemble test sets;
- select the testing levels (unit, integration, etc.)
- appropriate for the tests;
- evaluate the results of the tests

Defect Classes, the Defect Repository, and Test Design

- R e q u i r e m e n t s a n d S p e c i f i c a t i o n D e f e c t s
- D e s i g n D e f e c t s
- C o d i n g D e f e c t s
- T e s t i n g D e f e c t s

R e q u i r e m e n t s a n d S p e c i f i c a t i o n D e f e c t s

- Functional Description Defects
- Feature Defects
- Feature Interaction Defects
- Interface Description Defects

D e s i g n D e f e c t s

- Algorithmic and Processing Defects
- Control, Logic, and Sequence Defects
- Data Defects
- Module Interface Description Defects
- Functional Description Defects

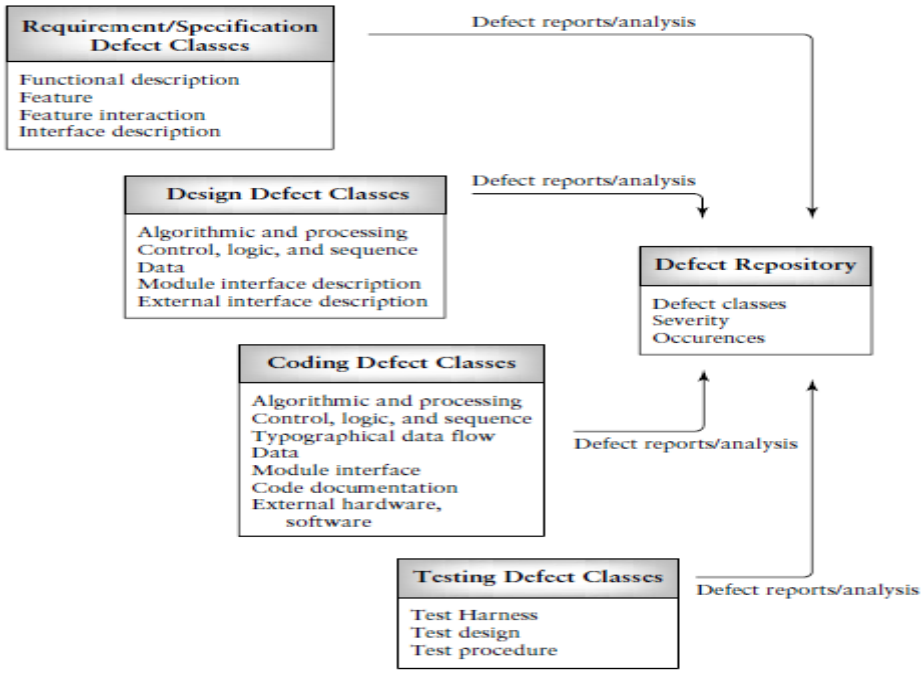
- External Interface Description Defects

C o d i n g D e f e c t s

- Algorithmic and Processing Defects
- Control, Logic and Sequence Defects
- Typographical Defects
- I n i t i a l i z a t i o n Defects
- Data-Flow Defects
- Data Defects
- Module Interface Defects
- Code Documentation Defects
- External Hardware, Software Interfaces Defects

T e s t i n g D e f e c t s

- Test Harness Defects
- Test Case Design and Test Procedure Defects



Defect Examples: The Coin Problem

- A precondition is a condition that must be true in order for a software component to operate properly.
- In this case a useful precondition would be one that states for example: number_of_coins >0
- **A postcondition is a condition that must be true when a software component completes its operation properly.**

- A useful postcondition would be:
number_of_dollars, number_of_cents>= 0.

Specification for Program calculate_coin_value

This program calculates the total dollars and cents value for a set of coins. The user inputs the amount of pennies, nickels, dimes, quarters, half-dollars, and dollar coins held. There are six different denominations of coins. The program outputs the total dollar and cent values of the coins to the user.

Inputs: number_of_coins is an integer
Outputs: number_of_dollars is an integer
number_of_cents is an integer

A sample specification with defects.

```
Design Description for Program calculate_coin_values

Program calculate_coin_values
number_of_coins is  integer
total_coin_value is integer
number_of_dollars is integer
number_of_cents is integer
coin_values is array of  six integers representing
each coin value in cents
initialized to: 1,5,10,25,25,100
begin

initialize total_coin_value to zero
initialize loop_counter to one
while loop_counter is less then six
begin
        output "enter number of coins"
        read (number_of_coins )
        total_coin_value = total_coin_value +
        number_of_coins * coin_value[loop_counter]
        increment loop_counter
end
number_dollars =  total_coin_value/100
number_of_cents = total_coin_value - 100 * number_of_dollars
output (number_of_dollars, number_of_cents)
end
```

A sample design specification with defects.

- **Control, logic, and sequencing defects**. The defect in this subclass arises from an incorrect "while" loop condition (should be less than or equal to six)
- **Data defects.** This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read1,5,10,25,50,100.

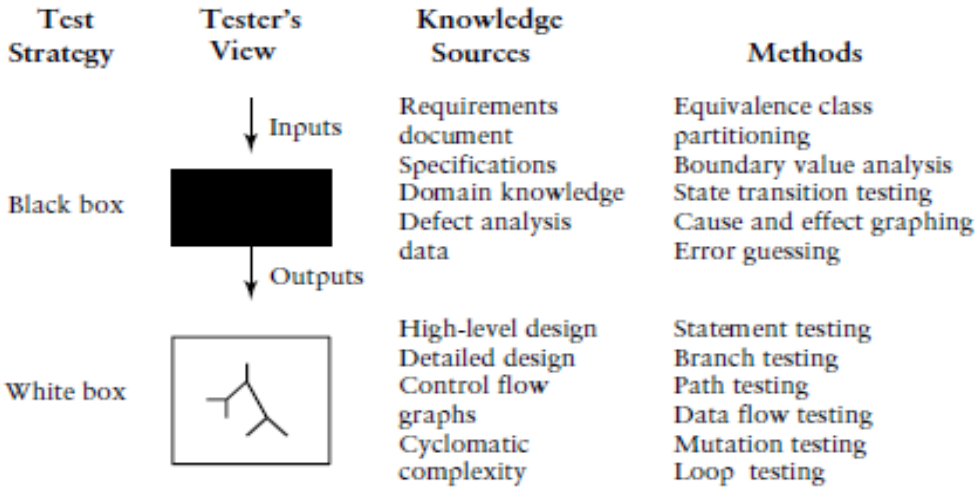
```
/*********************************************************************
program calculate_coin_values  calculates the dollar and cents
value of a set of coins of different dominations input by the user
denominations are pennies, nickels, dimes, quarters, half dollars,
and dollars
*********************************************************************/
main ()
{
int total_coin_value;
int number_of_coins = 0;
int number_of_dollars = 0;
int number_of-cents = 0;
int coin_values = {1,5,10,25,25,100};
{
int  i = 1;
while ( i < 6)
{
       printf("input number of coins\n");
       scanf ("%d", number_of_coins);
       total_coin_value = total_coin_value +
        (number_of_coins * coin_value{i]);
}
i = i + 1;
number_of_dollars =  total_coin_value/100;
number_of_cents = total_coin_value - (100 * number_of_dollars);
printf("%d\n", number_of_dollars);
printf("%d\n", number_of-cents);
}

/*********************************************************************/
```

A code example with defect

Unit-II Test Case Design

Strategies and Methods for Test Case Design –I

Test Case Design Strategies



The two basic testing strategies

Using the Black Box Approach to Test Case Design

- black box test strategy where we are considering only inputs and outputs as a basis for designing test cases
- how do we choose a suitable set of inputs from the set of all possible valid and invalid inputs
- As a example, suppose you tried to test a single procedure that calculates the square root of a number.
- If you were to exhaustively test it you would have to try all positive input values.
- what about all negative numbers, fractions? These are also possible inputs.

2.1 Equivalence Class Partitioning

- A good approach to selecting test inputs is to use a method called equivalence class partitioning.
- advantages:
  - It eliminates the need for exhaustive testing, which is not feasible.
  - It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.
  - It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

There are several important points related to equivalence class partitioning

- The tester must consider both valid and invalid equivalence classes.
- Equivalence classes may also be selected for output conditions.
- The derivation of input or outputs equivalence classes is a heuristic process.

List of Conditions

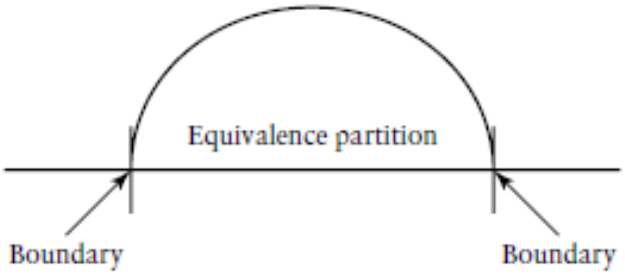
- If an input condition for the software-under-test is specified as a range of values
- 'If an input condition for the software-under-test is specified as a *number* of values

- If an input condition for the software-under-test is specified as a *set* of valid input values
- ‘‘If an input condition for the software-under-test is specified as a “*must be*” condition

```
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
```

- 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.

2.2 Boundary Value Analysis



Boundaries of an equivalence partition

An Example of the Application of Equivalence Class Partitioning and Boundary Value Analysis

- Widget identifiers into a widget data base
- We have three separate 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,
(iii) the first two characters must be letters.

- First we consider condition 1, the requirement for alphanumeric characters. This is a “must be” condition. We derive two equivalence classes.
  - EC1. Part name is alphanumeric, valid.
  - EC2. Part name is not alphanumeric, invalid.
- Then we treat condition 2, the range of allowed characters 3–15.
  - 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.

- Finally we treat the "must be" case for the first two characters.
  - EC6. The first 2 characters are letters, valid.
  - EC7. The first 2 characters are not letters, invalid.

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

Fig 2.3 Example equivalence class reporting table

- For example:
  - **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

Module name: Insert_Widget
Module identifier: AP62-Mod4
Date: January 31, 2000
Tester: Michelle Jordan

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)	

Summary of test inputs using equivalence class partitioning and boundary value analysis for sample module

- A major weakness with equivalence class partitioning is that it does not allow testers to combine conditions.

2.3 Cause-and-Effect Graphing

- The tester must decompose the specification of a complex software component into lower-level units.
- For each specification unit, the tester needs to identify causes and their effects.
- From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects.
- The graph may be annotated with constraints that describe combinations of causes and/or effects that are not possible due to environmental or syntactic constraints.
- The graph is then converted to a decision table.
- The columns in the decision table are transformed into test cases.

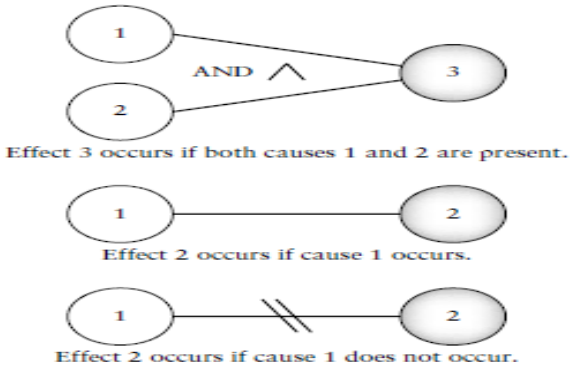
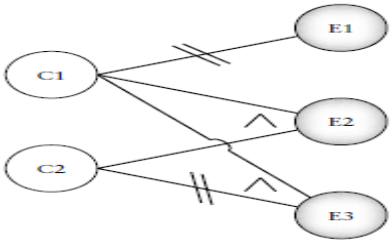


Fig 2.5 Sample of cause-and-effect graph notations

- The input conditions, or causes are as follows:

C1: Positive integer from 1 to 80
C2: Character to search for is in string
The output conditions, or effects are:
E1: Integer out of range
E2: Position of character in string
E3: Character not found
The rules or relationships can be described as follows:
If C1 and C2, then E2.
If C1 and not C2, then E3.
If not C1, then E1.



Cause-and-effect graph for the character search example

- A decision table will have a row for each cause and each effect.
- Entries in the table can be represented by a "1" for a cause or effect that is present, a "0" represents the absence of a cause or effect, and a "—" indicates a "don't care" value.
- A decision table for our simple example is shown in Table 4.3 where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases.

Inputs	Length	Character to search for	Outputs
T1	5	c	3
T2	5	w	Not found
T3	90		Integer out of range

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

Decision table for character search example

2.4 State Transition Testing

- A state is an internal configuration of a system or component. It is defined in terms of the values assumed at a particular time for the variables that characterize the system or component.
- A finite-state machine is an abstract machine that can be represented by a state graph having a finite number of states and a finite number of transitions between states.

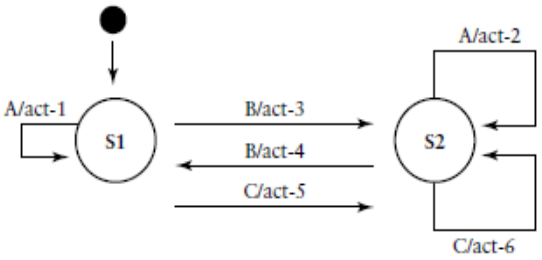


Fig. 2.8 Simple state transition graph

- For example, the transition from S1 to S2 occurs with input, or event B. Action 3 occurs as part of this state transition. This is represented by the symbol "B/act3.
- For the simple state machine in Figure 4.6 and Table 4.4 the transitions to be tested are:
    - Input A in S1
    - Input A in S2
    - Input B in S1
    - Input B in S2
    - Input C in S1
    - Input C in S2

	S1	S2
Inputs		
Input A	S1 (act-1)	S2 (act-2)
Input B	S2 (act-3)	S1 (act-4)
Input C	S2 (act-5)	S2 (act-6)

Fig 2.9 A state table for the machine

2.5 E r r o r G u e s s i n g

- Designing test cases using the error guessing approach is based on the tester's/developer's past experience

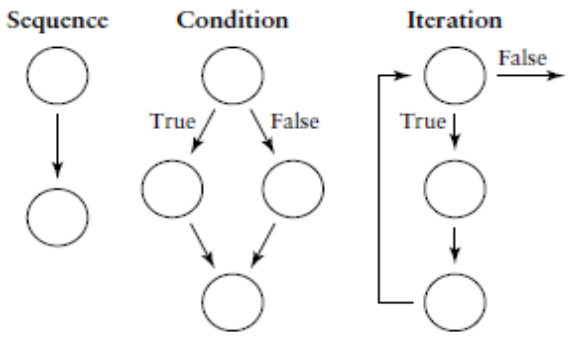
Strategies and Methods for Test Case Design –II

Using the White Box Approach to Test Case Design

- 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.

2.6 Coverage and Control Flow Graphs

- program statements
- decisions/branches (these influence the program flow of control)
- conditions (expressions that evaluate to true/false, and do not contain any other true/false-valued expressions)
- combinations of decisions and conditions
- paths (node sequences in flow graphs)
- All structured programs can be built from three basic primes-sequential (e.g., assignment statements), decision (e.g., if/then/else statements), and iterative (e.g., while, for loops).

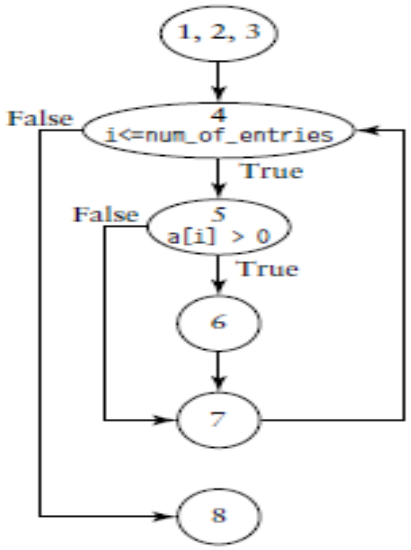


Representation of program primes

Covering Code Logic

```
/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer
array a. Input parameters are num_of_entries, an integer, and a, an array of integers with
num_of_entries elements. The output parameter is the integer sume */

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

Code sample with branch and loop



A control flow graph representation for the code

Decision or branch	Value of variable i	Value of predicate	Test case: Value of a, num_of_entries
			a=1, −45,3 num_of_entries = 3
while	1	True	
	4	False	
if	1	True	
	2	False	

A test case for the code in 2.11 that satisfied the decision coverage criterion

```
if(age <65 and married  true)
do X
do Y .....
else
do Z
```

- Condition 1: Age less than 65
- Condition 2: Married is true

Test cases for simple decision coverage

Value for age	Value for married	Decision outcome (compound predicate as a whole)	Test case ID
30	True	True	1
75	True	False	2

Test cases for condition coverage

Value for age	Value for married	Condition 1 outcome	Condition 2 outcome	Test case ID
75	True	False	True	2
30	False	True	False	3

2.7 Paths: Their Role in White Box–Based Test Design

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

1-2-3-4-8

8 White Box Test Design Approaches

2.8.1 Data Flow and White Box Test Design

We say a variable is defined in a statement when its value is assigned or changed.

- For example in the statements

Y= 26 * X
Read (Y)

  - the variable *Y* is defined, that is, it is assigned a new value. In data flow notation this is indicated as a *def* for the variable *Y*.

We say a variable is used in a statement when its value is utilized in a statement. The value of the variable is not changed.

- They describe a predicate use (p-use) for a variable that indicates its role in a predicate. A computational use (c-use) indicates the variable's role as a part of a computation. In both cases the variable value is unchanged.

- For example, in the statement

  Y = 26 * X
- the variable X is used. Specifically it has a c-use.
- In the statement

if (X > 98)
Y = max

- X has a predicate or p-use.

1	sum = 0	*sum, def*
2	read (n),	*n, def*
3	i = 1	*i, def*
4	while (i <= n)	*i, n p-sue*
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*

Fig 2.14 Sample code with data flow information

2.8.2 L o o p T e s t i n g

- Loop testing strategies focus on detecting common defects associated with these structures.

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely;
(ii) one iteration of the loop;
(iii) two iterations of the loop;
(iv) k iterations of the loop where k <n;

(v) n - 1 iterations of the loop;
(vi) n + 1 iterations of the loop (if possible).

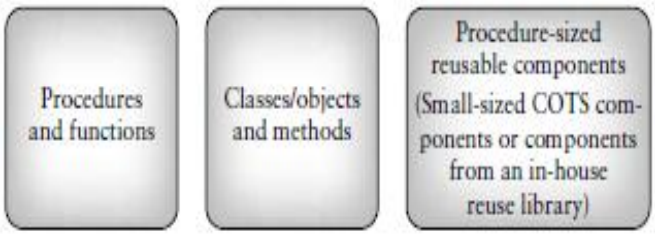
Unit-III
Levels of Testing

3.1 Unit Test: Functions, Procedures, Classes, and Methods as Units

- A unit is the smallest possible testable software component.

It can be characterized in several ways. For example, a unit in a typical

- procedure-oriented software system:
- performs a single cohesive function;
- can be compiled separately;
- is a task in a work breakdown structure (from the manager's point of view);
- contains code that can fit on a single page or screen.



some components suitable for unit test

Unit Test: The Need for Preparation

- To prepare for unit test the developer/tester must perform several tasks. These are:

(i) plan the general approach to unit testing;
(ii) design the test cases, and test procedures (these will be attached to the test plan);
(iii) define relationships between the tests;
(iv) prepare the auxiliary code necessary for unit test.

Unit Test Planning

- **Phase 1: Describe Unit Test Approach and Risks**

  In this phase of unit testing planning the general approach to unit testing is outlined. The test planner:.

  (i) identifies test risks;
  (ii) describes techniques to be used for designing the test cases for the units;
  (iii) describes techniques to be used for data validation and recording of test results;
  (iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.

- **Phase 2: Identify Unit Features to be Tested**
- **Phase 3: Add Levels of Detail to the Plan**

Designing the Unit Tests

- Part of the preparation work for unit test involves unit test design. It is important to specify

(i) the test cases (including input data, and expected outputs for each test case), and,
(ii) the test procedures (steps required run the tests)

The Class as a Testable Unit: Special Considerations

- **Issue 1: Adequately Testing Classes**

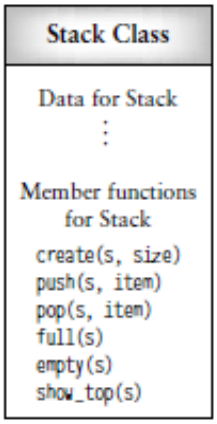


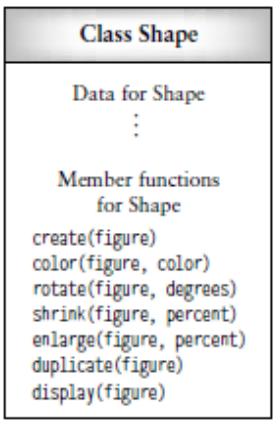
Fig 3.2 Sample stack class with multiple methods

Issue 2: Observation of Object States and State Changes

- empty(s), push(s,item-1), show_top(s), push(s,item-2),
- show_top(s), push(s,item-3), full(s), show_top(s), pop(s,item),
- show_top(s), pop(s,item), show_top(s), empty(s), . . .

Issue 3: The Retesting of Classes—I

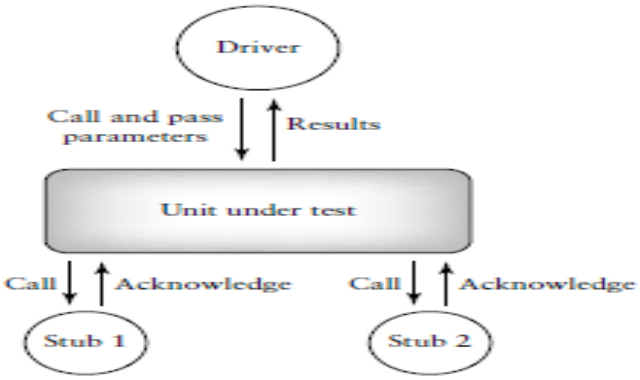
Issue 4: The Retesting of Classes—II



Sample shape class

The Test Harness

- The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls.



The test harness

For example, a driver could have the following options and combinations of options:
(i) call the target unit;
(ii) do 1, and pass inputs parameters from a table;
(iii) do 1, 2, and display parameters;
(iv) do 1, 2, 3 and display results (output parameters).

The stubs could also exhibit different levels of functionality. For example a stub could:

(i) display a message that it has been called by the target unit;
(ii) do 1, and display any input parameters passed from the target unit;
(iii) do 1, 2, and pass back a result from a table;
(iv) do 1, 2, 3, and display result from table.

Running the Unit Tests and Recording Results

Unit Test Worksheet

Unit Name: _____

Unit Identifier: _____

Tester: _____

Date: _____

Test case ID	Status (run/not run)	Summary of results	Pass/fail

Summary work sheet for unit test

3.2 Integration Test: Goals

- Integration test for procedural code has two major goals:
  (i) to detect defects that occur on the interfaces of units;
  (ii) to assemble the individual units into working subsystems and finally a complete system that is ready for system test.

Integration Strategies for Procedures and Functions

- For conventional procedural/functional-oriented systems there are two major integration strategies—top-down and bottom-up.

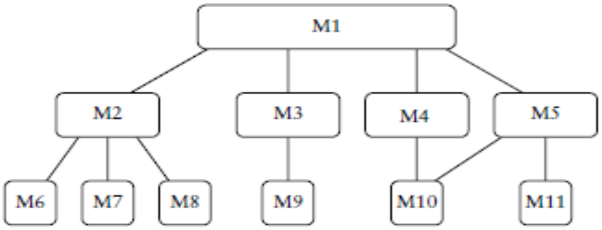


Fig 3.6 Simple structure chart for integration test examples

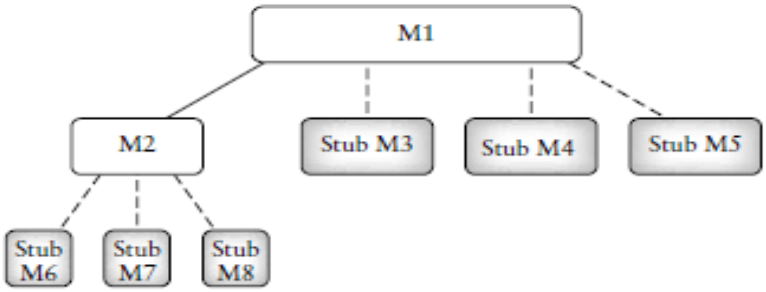
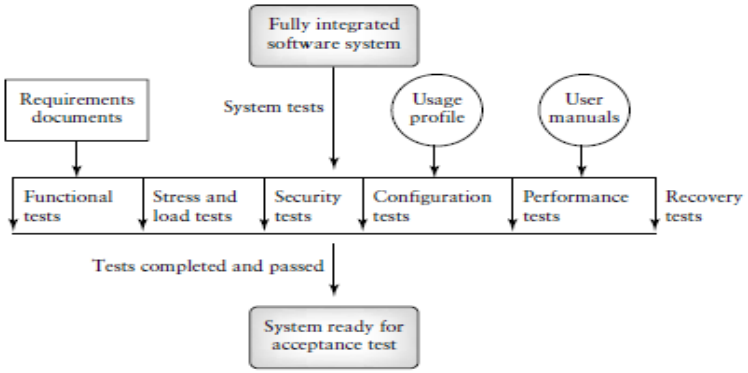


Fig 3.7 Top-down integration of modules M1 and M2

System Test: The Different Types

There are several types of system tests as shown on Figure. The types are as follows:

• Functional testing
• Performance testing
• Stress testing
• Configuration testing
• Security testing
• Recovery testing



Types of system test

Functional Testing

- All types or classes of legal inputs must be accepted by the software.
- All classes of illegal inputs must be rejected (however, the system should remain available).
- All possible classes of system output must exercised and examined.
- All effective system states and state transitions must be exercised and examined.
- All functions must be exercised.

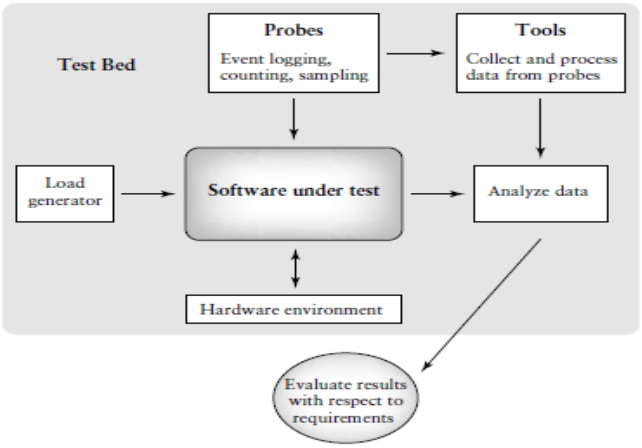
Performance Testing

1. *Functional requirements.*

Users describe what functions the software should perform. We test for compliance of these requirements at the system level with the functional-based system tests.

2. *Quality requirements.*

There are nonfunctional in nature but describe quality levels expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughput, and delays.



Example of special resources needed for a performance test

Stress Testing

- When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing.

- For example, if an operating system is required to handle 10 interrupts / second and the load causes 20 interrupts / second, the system is being stressed. The goal of stress test is to try to break the system; find the circumstances under which it will crash. This is sometimes called “breaking the system.”

C o n f i g u r a t i o n T e s t i n g

- Typical software systems interact with hardware devices such as disc drives, tape drives, and printers.

Configuration testing has the following objectives

• Show that all the configuration changing commands and menus work properly.
• Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.
• Show that the systems' performance level is maintained when devices are interchanged, or when they fail.

S e c u r i t y T e s t i n g

- Designing and testing software systems to insure that they are safe and secure is a big issue facing software developers and test specialists.

Computer software and data can be compromised by:

i) criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy;
(ii) errors on the part of honest developers/maintainers who modify, destroy**,** or compromise data because of is information, misunderstandings, and/or lack of knowledge.

Damage can be done through various means such as:
(i) viruses;
(ii) trojan horses;
(iii) trap doors;
(iv) illicit channels.

- The effects of security breaches could be extensive and can cause:

(i) loss of information;
(ii) corruption of information;
(iii) misinformation;
(iv) privacy violations;
(v) denial of service.

R e c o v e r y T e s t i n g

- Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses.
- This type of testing is especially important for transaction systems, for example, on-line banking software.

Testers focus on the following areas during recovery testing

- ***Restart.*** *The current system state and transaction states are discarded.* The most recent checkpoint record is retrieved and the system initialized to the states in the checkpoint record. Testers must insure that all transactions have been

reconstructed correctly and that all devices are in the proper state. The system should then be able to begin to process new transactions.

- ***Switchover.*** *The ability of the system to switch to a new processor* must be tested. Switchover is the result of a command or a detection of a faulty processor by a monitor.

- In each of these testing situations all transactions and processes must be carefully examined to detect:

(i) loss of transactions;
(ii) merging of transactions;
(iii) incorrect transactions;
(iv) an unnecessary duplication of a transaction.

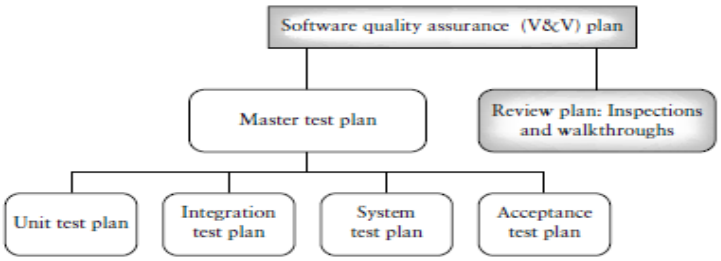
Regression Testing

- Regression testing is not a level of testing, but it is the *retesting of software* that occurs when changes are made to ensure that the new version of the software

Unit -IV

Test Management

- A plan is a document that provides a framework or approach for achieving a set of goals.
- Milestones are tangible events that are expected to occur at a certain time in the project's lifetime. Managers use them to determine project status
- Test plans for software projects are very complex and detailed documents. The planner usually includes the following essential high-level items.
- *Overall test objectives*
    - *What to test (scope of the tests).*
    - *Who will test.*
    - *How to test.*
    - *When to test.*
    - *When to stop testing*



Test Plan Components

Test Plan Components

1. Test plan identifier
2. Introduction
3. Items to be tested
4. Features to be tested
5. Approach
6. Pass/fail criteria
7. Suspension and resumption criteria
8. Test deliverables
9. Testing Tasks
10. Test environment
11. Responsibilities
12. Staffing and training needs
13. Scheduling
14. Risks and contingencies
15. Testing costs
16. Approvals

Components of a test plan

4.0 Test Planning

4.1 Meet with project manager. Discuss test requirements.
4.2 Meet with SQA group, client group. Discuss quality goals and plans.
4.3 Identify constraints and risks of testing.
4.4 Develop goals and objectives for testing. Define scope.
4.5 Select test team.
4.6 Decide on training required.
4.7 Meet with test team to discuss test strategies, test approach, test monitoring, and controlling mechanisms.
4.8 Develop the test plan document.
4.9 Develop test plan attachments (test cases, test procedures, test scripts).
4.10 Assign roles and responsibilities.
4.11 Meet with SQA, project manager, test team, and clients to review test plan.

A breakdown of testing planning element

Test Plan Attachments

Requirement identifier	Requirement description	Priority (scale 1–10)	Review status	Test ID
SR-25-13.5	Displays opening screens	8	Yes	TC-25-2 TC-25-5
SR-25-52.2	Checks the validity of user password	9	Yes	TC-25-18 TC-25-23

Example of entries in a requirements traceability matrix

Test Design Specification

- *Test Design Specification Identifier*
- *Features to Be Tested*
- *Approach Refinements*
- *Test Case Identification*
- *Pass/Fail Criteria*

Test Case specification

- *Test Case Specification Identifier*
- *Test Items*
- *Input Specifications*
- *Output Specifications*
- *Special Environmental Needs*
- *Special Procedural Requirements*
- *Intercase Dependencies*

Test procedure specification

- A procedure in general is a sequence of steps required to carry out a specific task.
- *Test Procedure Specification Identifier*
- *Purpose*
- *Specific Requirements*
- *Procedure Steps*

i) setup: to prepare for execution of the procedure;

(ii) start: to begin execution of the procedure;

proceed: to continue the execution of the procedure;

(iv) measure: to describe how test measurements related to outputs will

be made;

(v) shut down: to describe actions needed to suspend the test when unexpected

events occur;

(vi) restart: to describe restart points and actions needed to restart the

procedure from these points;

(vii) stop: to describe actions needed to bring the procedure to an orderly

halt;

(viii) wrap up: to describe actions necessary to restore the environment;

(ix) contingencies: plans for handling anomalous events if they occur

during execution of this procedure.

Locating Test Items: The Test Item Transmittal Report

(i) version/revision number of the item;

(ii) location of the item;

(iii) persons responsible for the item (e.g., the developer);

(iv) references to item documentation and the test plan it is related to;

(v) status of the item;

(vi) approvals—space for signatures of staff who approve the transmittal.

Reporting Test Results

- **Test Log**
  - *Test Log Identifier*
  - *Description*
  - *Activity and Event Entries*
    - *Execution description*
    - *Procedure results*
    - *Environmental information*
    - *Anomalous events*
    - *Incident report identifiers*
- **Test Incident Report**

1. *Test Incident Report identifier: to uniquely identify this report.*

2. *Summary: to identify the test items involved, the test procedures, test*

cases, and test log associated with this report.

3. *Incident description: this should describe time and date, testers, observers,*

environment, inputs, expected outputs, actual outputs, anomalies, procedure step, environment, and attempts to repeat.

4. *Impact: what impact will this incident have on the testing effort,* *the* test plans, the test procedures, and the test cases

- **Test Summary Report**

1. *Test Summary Report identifier: to uniquely identify this report.*

2. *Variances: these are descriptions of any variances of the test items* from their original design.

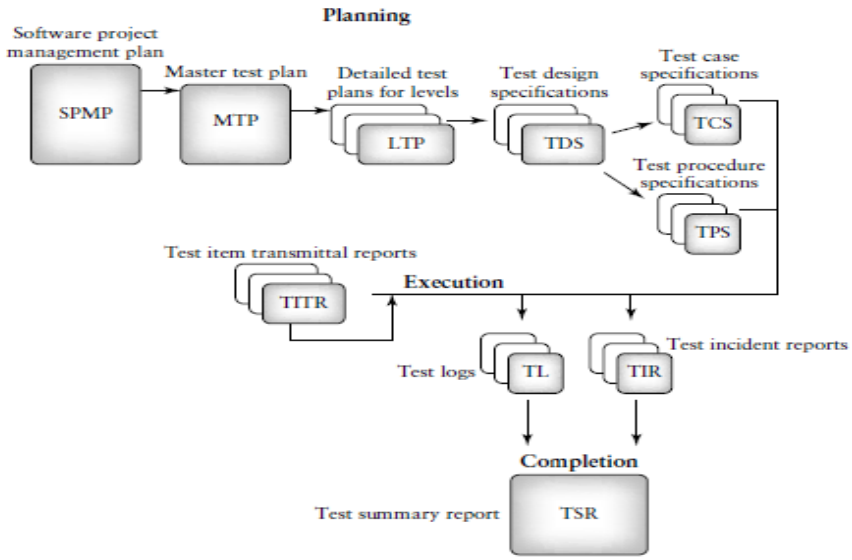
3. Comprehensiveness assessment: the document author discusses the comprehensiveness of the test effort as compared to test objectives

Summary of results: the document author summarizes the testing results.

5. *Evaluation: in this section the author evaluates each test item based* on test results.

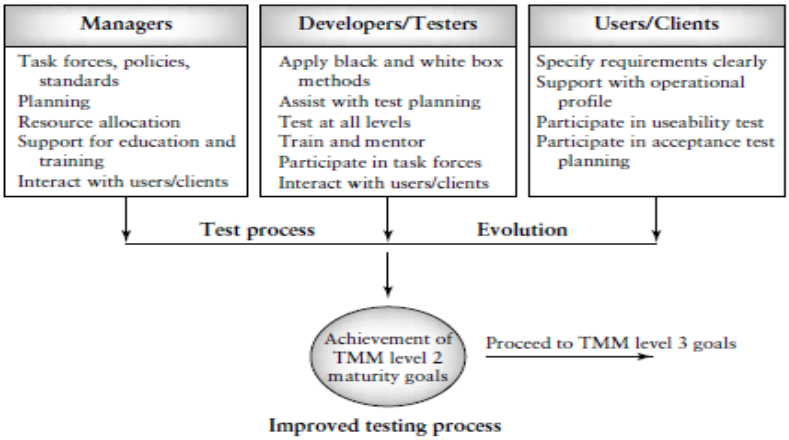
6. *Summary of activities: all testing activities and events are summarized.*

7. *Approvals: the names of all persons who are needed to approve this* document are listed with space for signatures and dates.



Test-related documents as recommended by IEEE[5]

The Role of the Three Critical Groups in Testing Planning and Test Policy Development



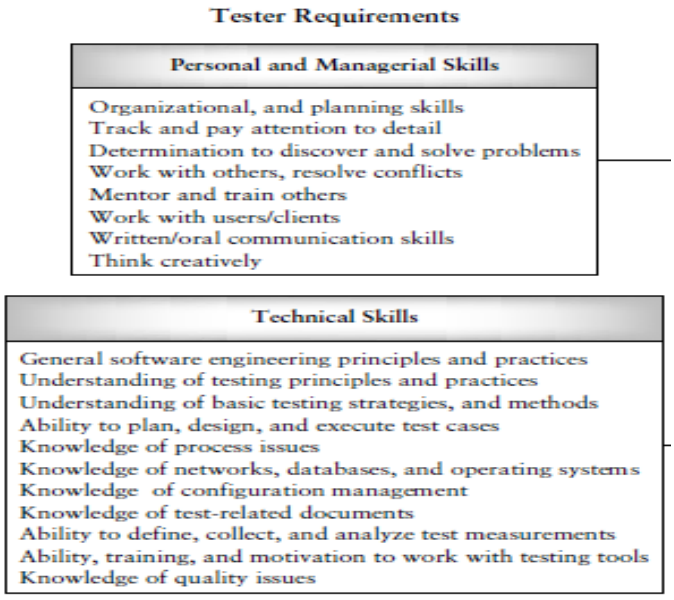
Reaching TMM level 2; summary of critical group roles

Introducing the Test Specialist

- maintenance and application of test policies;
- development and application of test-related standards;
- participating in requirements, design, and code reviews;
- test planning;
- test design;
- test execution;
- test measurement;
- test monitoring (tasks, schedules, and costs);
- defect tracking, and maintaining the defect repository;
- acquisition of test tools and equipment;
- identifying and applying new testing techniques, tools, and methodologies;
- mentoring and training of new test personnel;
- test reporting.

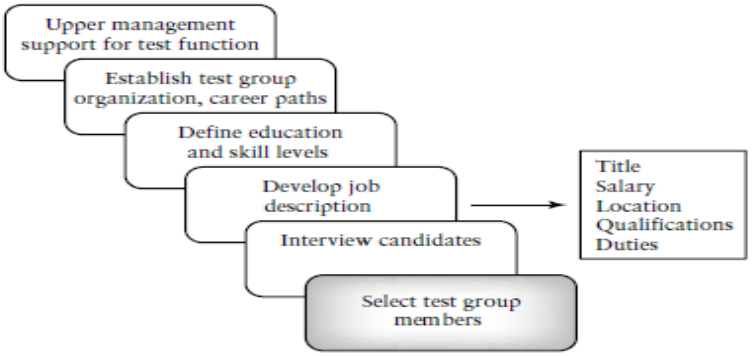
Skills Needed by a Test Specialist

- organizational, and planning skills;
- the ability to keep track of, and pay attention to, details;
- the determination to discover and solve problems;
- the ability to work with others and be able to resolve conflicts;
- the ability to mentor and train others;
- the ability to work with users and clients;
- strong written and oral communication skills;
- the ability to work in a variety of environments;
- the ability to think creatively



Test specialist skills

Building a Testing Group



Test specialist skills

The Structure of the Test Group

• maintain testing policy statements;

• plan the testing efforts;

• monitor and track testing efforts so that they are on time and within budget;

• measure process and product attributes;

• provide management with independent product and process quality information;

• design and execute tests with no duplication of effort;

• automate testing;

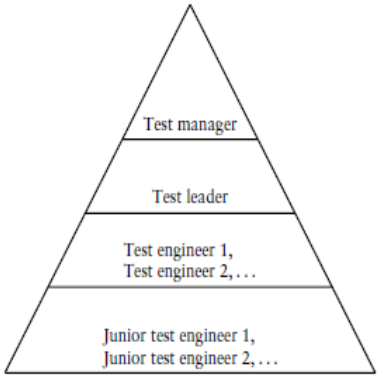
• participate in reviews to insure quality;

• work with analysts, designers, coders, and clients to ensure quality goals are meet;

• maintain a repository of test-related information;

• give greater visibility to quality issues organization wide;

• support process improvement efforts.



The test team hierarchy

The Technical Training Program

• quality issues;

• measurement identification, collection, and analysis;

• testing techniques and methodologies;

• design techniques;

• tool usage (for all life cycle phases);

• configuration management;

• planning;

• process evaluation and improvement;

• policy development;

• technical review skills;

• software acquisition;

• project management skills;

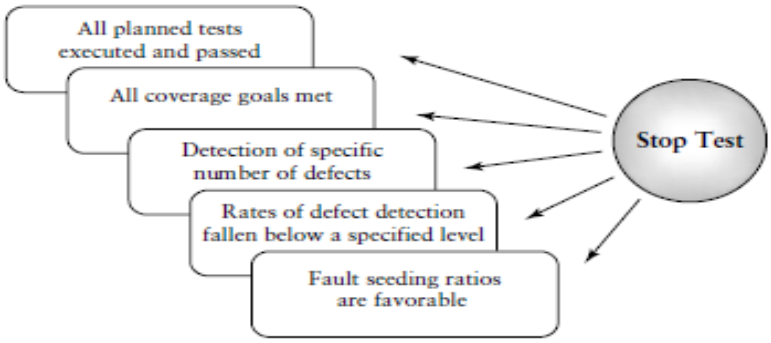
• business skills

• communication skills.

Unit -V Test Automation

Status Meetings, Reports, and Control Issues

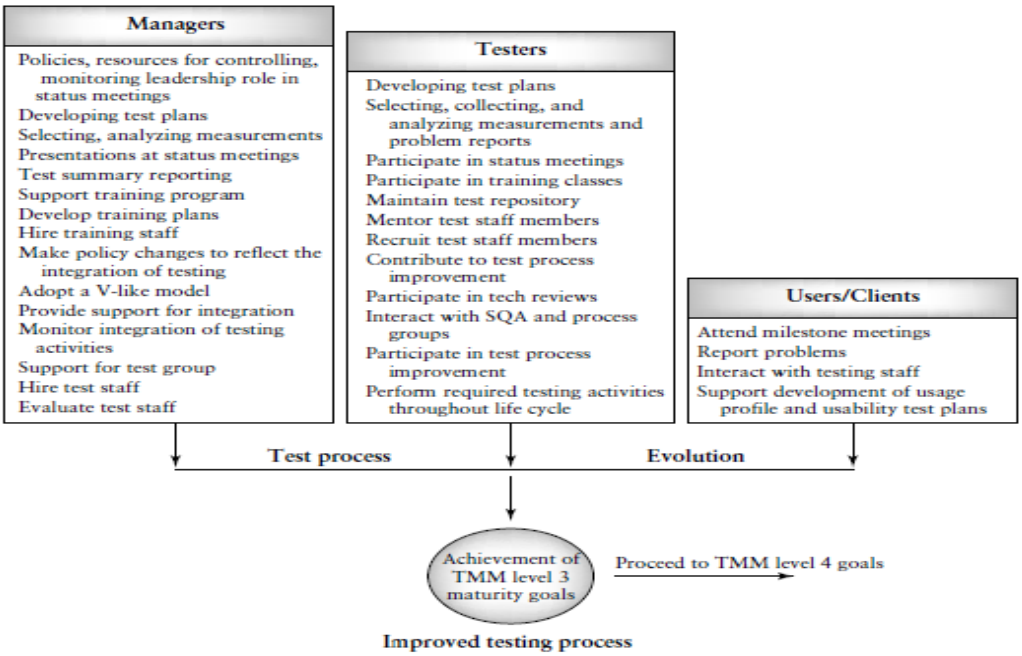
- All the Planned Tests That Were Developed Have Been Executed and Passed.
- All Specified Coverage Goals Have Been Met.
- The Detection of a Specific Number of Defects Has Been Accomplished.
- The Rates of Defect Detection for a Certain Time Period Have Fallen Below a Specified Level.
- Fault Seeding Ratios Are Favorable



Software Configuration Management

- **Identification of the Configuration Items**
- **Change Control**
- **Configuration status reporting**
- **Configuration audits**

Controlling and Monitoring: Three Critical Views



Contributions of three critical groups to TMM level 3 maturity goals

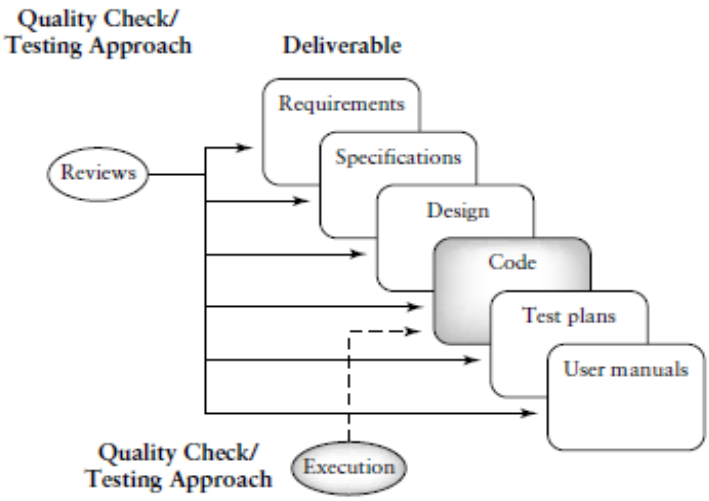
REVIEWS AS ATESTING ACTIVITY

i) testing policies with an emphasis on defect detection and quality, and measurements for controlling and monitoring;

(ii) a test organization with staff devoted to defect detection and quality issues;

(iii) policies and standards that define requirements, design, test plan, and other documents;

(iv) organizational culture with a focus on quality products and quality processes.



Role of reviews in testing software deliverables

The many benefits of a review program are:

- higher-quality software;
- increased productivity (shorter rework time);
- closer adherence to project schedules (improved process control);
- increased awareness of quality issues;
- teaching tool for junior staff;
- opportunity to identify reusable software artifacts;
- reduced maintenance costs;
- higher customer satisfaction;
- more effective test planning;
- a more professional attitude on the part of the development staff.

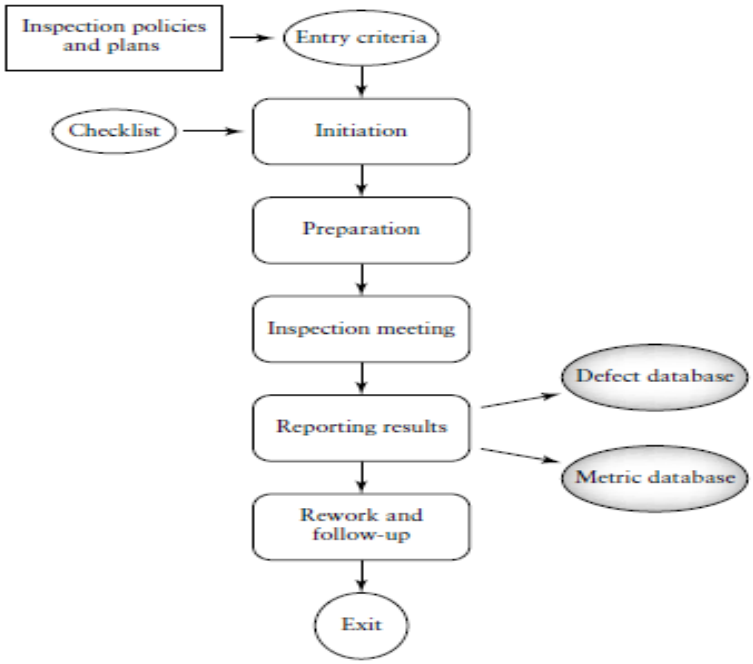
Types of Reviews

- verify that a software artifact meets its specification;
- to detect defects; and
- check for compliance to standards.

- Inspections as a Type of Technical Review

- Walkthroughs as a Type of

Technical Review



Steps in the inspection process

The Need for Review Policies

i) testing policies with an emphasis on defect detection and quality, and measurements for controlling and monitoring;

(ii) a test organization with staff devoted to defect detection and quality issues;

(iii) policies and standards that define requirements, design, test plan, and other documents;

(iv) organizational culture with a focus on quality products and quality processes.

EVALUATING SOFTWAREQUALITY: AQUANTITATIVEAPPROACH

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.

2. Quality relates to the degree to which a system, system component, or process, meets customer, or user, needs or expectations.

The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software conforms to established technical requirements.

Quality Costs

• quality planning;

• test and laboratory equipment;

• training;

• formal technical reviews.

What Is Quality Control?

- Quality control consists of the procedures and practices employed to ensure that a work product or deliverable conforms to standards or requirements.
- Quality control is the set of activities designed to evaluate the quality of developed or manufactured products.

Quality control includes:

• policies and standards;

• review and audit procedures;

• a training program;

• dedicated, trained, and motivated staff (for testing and quality assurance);

• a measurement program;

• a planning process (for test and quality assurance);

• effective testing techniques, statistical techniques, and tools;

• process monitoring and controlling systems;

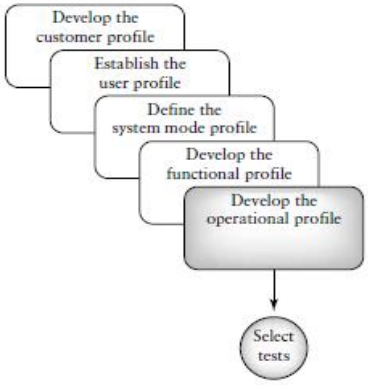
• a test process assessment system (TMM-AM);

• a configuration management system.

The Role of Operational Profiles and Usage Models in Quality Control

1. An operational profile is a quantitative characterization of how a software system will be used in its intended environment .

2. An operational profile is a specification of classes of inputs and the probability of their occurrence.



Steps to develop an operations profile

Software Reliability

- Software reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time .
- Software reliability is the probability that a software system will operate without failure under given conditions for a given time interval.
- Availability is the degree to which a system or component is operational and accessible when required for use .
- Availability is the probability that a software system will be available for use.
- Trustworthiness is the probability that there are no errors in the software that will cause the system to fail catastrophically.

Measurements for Software Reliability

- MTBF = MTTF + MTTR
- mean time to failure, MTTF

- mean time to repair, MTTR
- mean time between failure, MTBF
- Reliability (R)

R=MTBF/ 1 + MTBF

- Availability (A)

A= MTBF / MTBF + MTTR

- The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changing environment.
- maintainability (M)

M= 1 / 1+ MTTR

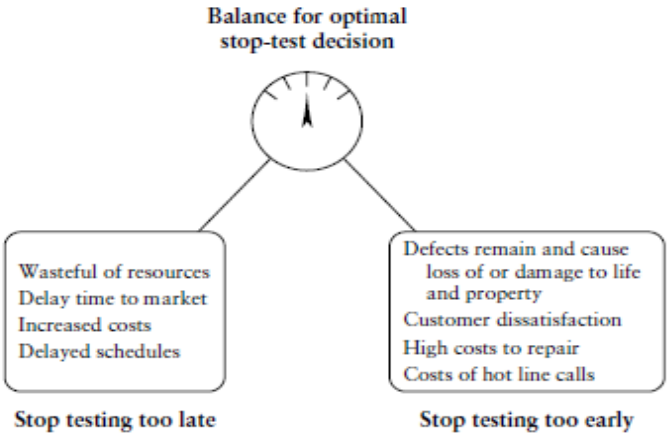
Reliability, Quality Control, and Stop-Test Decisions

(i) setting reliability goals in the requirements document (a reliability specification is the result);

(ii) providing adequate time and resources in the test plan to allow for developing/modifying the usage profile, running the tests, and collecting,

cataloging, and analyzing the reliability data;

(iii) developing a usage or operational profile that accurately models usage patterns;

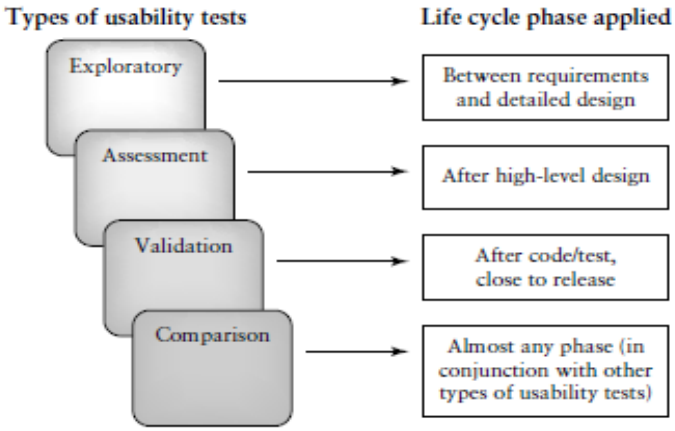


Consequences of untimely stop-test decisions

Usability Testing, and Quality Control

- Usability is a quality factor that is related to the effort needed to learn, operate, prepare input, and interpret the output of a computer program.
- Usability is a complex quality factor and can be decomposed according to IEEE standards into the sub factors
- Understandability: The amount of effort required to understand the software.
- Ease of learning: The degree to which user effort required to understand the software is minimized.
- Operability: The degree to which the operation of the software matches the purpose, environment, and physiological characteristics of users; this includes ergonomic factors such as color, shape, sound, font size, etc.
- Communicativeness: The degree to which the software is designed in accordance with the psychological characteristics of the users.

An Approach to Usability Testing



Types of usability tests.

Usability Testing: Resource Requirements

1. *A usability testing laboratory*

2. *Trained personnel*

3. *Usability test planning.*

Usability Tests and Measurements

(i) open an existing document;

(ii) add text to the document;

(iii) modify the old text;

(iv) change the margins in selected sections;

(v) change the font size in selected sections;

(vi) print the document;

(vii) save the document.