1.1 INTRODUCTION TO TESTING AS AN ENGINEERING ACTIVITY

This is an exciting time to be a software developer.

Because software now has such an important role in our lives both economically and socially, there is pressure for software professionals to focus on quality issues. Poor quality software that can cause loss of life or property is no longer acceptable to society. Failures can result in catastrophic losses.

Conditions demand software development staffs with interest and training in the areas of software product and process quality. Highly qualified staff ensure that software products are built on time, within budget, and are of the highest quality with respect to attributes such as reliability, correctness, usability, and the ability to meet all user requirements.

The profession of software engineering is slowly emerging as a formal engineering discipline. As a new discipline it will be related to other engineering disciplines, and have associated with it a defined body of knowledge, a code of ethics, and a certification process.

The education and training of engineers in each engineering discipline is based on the teaching of related *scientific principles, engineering processes, standards, methods, tools, measurement, code of ethics and best practices*.

The goals of the task force teams, appointed by IEEE Computer Society and ACM, are to define a body of knowledge that covers the software engineering discipline, to discuss the nature of education for this new profession, and to define a code of ethics for the software engineer. Foreseeing the emergence of this new engineering discipline, some states are already preparing licensing examinations for software engineers.

The aim of this subject is to support the education of a software professional called a test specialist. A test specialist is one whose education is based on the principles, practices, and processes that constitute the software engineering discipline, and whose specific focus is on one area of that discipline—software testing. A test specialist who is trained as an engineer should have knowledge of test-related principles, processes, measurements, standards, plans, tools, and methods, and should learn how to apply them to the testing tasks to be performed.

Using an engineering approach to software development implies that:

- 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;
- validation and verification processes play a key role in quality determination;
- Engineers have proper education, training, and certification.

1.2 ROLE OF PROCESS IN SOFTWARE QUALITY

Process in this context is defined below, and is illustrated in Figure 1.2.

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.

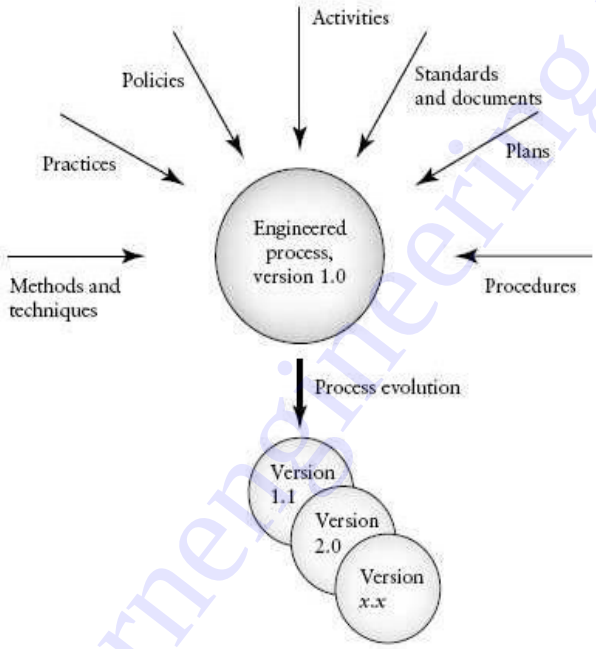


FIG. 1.2
Components of an engineered process.

A software process is a set of activities and associated results which produces a software product. These activities are,

1. Software specification → The functionality of software and constraints on its operation must be defined.
2. Software development → The software to meet the specification must be produced.
3. Software validation → The software must be validated to ensure that it does what the customer wants.
4. Software evolution → The software must evolve to meet changing customer needs.

The software development process, like most engineering artifacts, must be engineered. That is, it must be designed, implemented, evaluated, and maintained.

Engineering is the application of scientific, economic, social, and practical knowledge in order to design, build, and maintain structures, machines, devices, systems, materials and processes.

One who practices engineering is called an **engineer**, and those licensed to do so may have more formal designations such as **Professional Engineer**.

All the software process improvement models that have had wide acceptance in industry are high-level models, in the sense that they focus on the software process as a whole and do not offer adequate support to evaluate and improve specific software development sub processes such as design and testing.

In spite of its vital role in the production of quality software, existing process evaluation and improvement models such as the CMM, Bootstrap, and ISO-9000 have not adequately addressed testing process issues. The Testing Maturity Model (TMM), has been developed at the Illinois Institute of Technology by a research group, to address deficiencies these areas.

1.3 TESTING AS A PROCESS

Testing is an integral part of the software development process. Testing itself is related to two other processes called verification and validation as shown in Figure 1.3.Testing is not Validation or Verification.

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. Verification is usually associated with activities such as inspections and reviews of software deliverables.

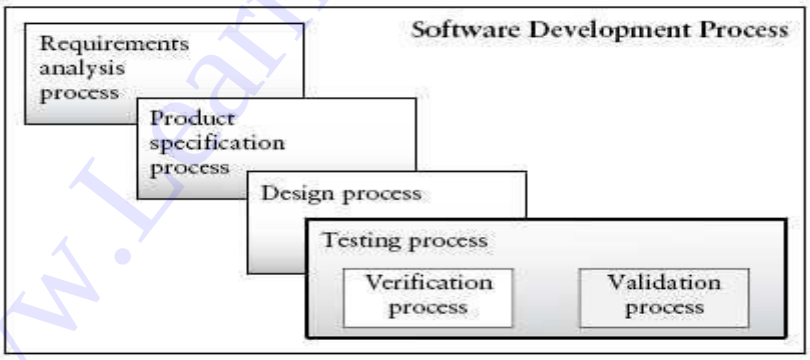


FIG. 1.3
Example processes embedded in the software development process.

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.

Testing is defined as a set of activities which are well planned in advance and also conducted systematically. It is a process of checking the software product in a predefined way to know if the behavior is same as the expected one.

Note that testing and debugging, or fault localization, are two very different activities. The debugging process begins *after* testing has been carried out and the tester has noted that the software is not behaving as specified.

Debugging, or fault localization is the process of (1) locating the fault or defect, (2) repairing the code, and (3) retesting the code.

Testing is a process, and as a process it must managed. Minimally that means that an organizational policy for testing must be defined and documented. Testing procedures and steps must be defined and documented. Testing must be planned, testers should be trained, the process should have associated quantifiable goals that can be measured and monitored. Testing as a process should be able to evolve to a level where there are mechanisms in place for making continuous improvements.

- Testing is a process, and as a process it must managed, that means that, an organizational policy for testing must be defined and documented.
- Testing procedures and steps must be defined and documented.
- Testing must be planned, testers should be trained,
- The process should have associated quantifiable goals that can be measured and monitored.
- Testing as a process should be able to evolve to a level where there are mechanisms in place for making continuous improvements

1.4 BASIC DEFINITIONS

Errors
An error is a mistake, misconception, or misunderstanding on the part of a software developer. In the category of developer we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a design notation, or a programmer might type a variable name incorrectly.

Faults (Defects)
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.
A fault in the code does not always produce a failure. In fact, faulty software may operate over a long period of time without exhibiting any incorrect behavior. However when the proper conditions occur the fault will manifest itself as a failure.

Test Cases

The usual approach to detecting defects in a piece of software is for the tester to select a set of input data and then execute the software with the input data under a particular set of conditions. In order to decide whether the software has passed or failed the test, the tester also needs to know, what are all the proper outputs for the software, given, the set of inputs and execution conditions. The tester bundles this information into an item called a test case.
A test case in a practical sense is a test-related item which contains the following information:

1. *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.
2. *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.
3. *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.**
- **customer or user needs, or expectations.**

We can measure the degree to which the software possess a given quality attribute with quality metrics.

- ✓ **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**

  - ❖ **correctness—**the degree to which the system performs its intended function
  - ❖ **reliability—**the degree to which the software is expected to perform its required functions under stated conditions for a stated period of time
  - ❖ **usability—**relates to the degree of effort needed to learn, operate, prepare input, and interpret output of the software

- **integrity—**relates to the system's ability to withstand both intentional and accidental attacks
- **portability—**relates to the ability of the software to be transferred from one environment to another
- **maintainability—**the effort needed to make changes in the software
- **interoperability—**the effort needed to link or couple one system to another.

Testability

The amount of effort needed to test the software to ensure it performs according to specified requirements (relates to number of test cases needed).

The ability of the software to reveal defects under testing conditions (some software is designed in such a way that defects are well hidden during ordinary testing conditions).

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 that the resulting software conforms to established technical requirements.

They work with project managers and testers to develop quality-related policies and quality assurance plans for each project. The group is also involved in measurement collection and analysis, record keeping, and reporting. The SQA team members participate in reviews and audits, record and track problems, and verify that corrections have been made. They also play a role in software configuration management.

1.5 SOFTWARE TESTING PRINCIPLES

Principle 1. Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.

Testers need to detect the defects before the software becomes operational. This principle supports testing as an execution-based activity to detect defects. It also supports the separation of testing from debugging since the intent of the latter is to locate defects and repair the software. The term ―software component is used in this context to represent any unit of software ranging in size and complexity from an individual procedure or method, to an entire software system. The term ―defects represents any deviations in the software that have a negative impact on its functionality, performance, reliability, security, etc.

Principle 2. When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yet undetected defect(s).

Principle 2 supports careful test design and provides a criterion with which to evaluate test case design and the effectiveness of the testing effort when the objective is to detect defects. It requires the tester to consider the goal for each test case, that is, which specific type of defect is to be detected by the test case. The goal for the test is to prove/disprove the hypothesis, that is, determine if the specific defect is present / absent. Based on the hypothesis, test inputs are selected, correct outputs are determined, and the test is run. Results are analyzed to prove/disprove the hypothesis.

Principle 3. Test results should be inspected meticulously.

Testers need to carefully inspect and interpret test results.

A failure may be overlooked, and the test may be granted a ―pass status when in reality the software has failed the test. Testing may continue based on erroneous test results. The defect may be revealed at some later stage of testing, but in that case it may be more costly and difficult to locate and repair.

A failure may be suspected when in reality none exists. In this case the test may be granted a ―fail status. Much time and effort may be spent on trying to find the defect that does not exist. A careful reexamination of the test results could finally indicate that no failure has occurred.

Principle 4. A test case must contain the expected output or result.

It is often obvious to the novice tester that test inputs must be part of a test case. However, the test case is of no value unless there is an explicit statement of the expected outputs or results. Expected outputs allow the tester to determine (i) whether a defect has been revealed, and (ii) pass/fail status for the test.

Principle 5. Test cases should be developed for both valid and invalid input conditions.

A tester must not assume that the software under test will always be provided with valid inputs. Inputs may be incorrect for several reasons. Software users often make typographical errors even when complete/correct information is available. Devices may also provide invalid inputs due to erroneous conditions and malfunctions. Use of test cases that are based on invalid inputs is very useful for revealing defects. Invalid inputs also help developers and testers evaluate the robustness of the software, that is, its ability to recover when unexpected events occur.

Principle 6. The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.

What this principle says is that the higher the number of defects already detected in a component, the more likely it is to have additional defects when it undergoes further testing. For example, if there are two components A and B, and testers have found 20 defects in A and 3 defects in B, then the probability of the existence of additional defects in A is higher than B. Defects often occur in clusters and often in code that has a high degree of complexity and is poorly designed.

Principle 7. Testing should be carried out by a group that is independent of the development group.

It is difficult for a developer to admit or conceive that software he/she has created and developed can be faulty. Testers must realize that developers, on a practical level, it may be difficult for them to conceptualize where defects could be found. Even when tests fail, developers often have difficulty in locating the defects since their mental model of the code may overshadow their view of code as it exists in actuality. The testing group could be implemented as a completely separate functional entity in the organization. As a member of any of these groups, the principal duties and training of the testers should lie in testing rather than in development. The groups need to cooperate so that software of the highest quality is released to the customer.

Principle 8. Tests must be repeatable and reusable.
This principle calls for experiments in the testing domain to require recording of the exact conditions of the test, any special events that occurred, equipment used, and a careful accounting of the results. This information is invaluable to the developers when the code is returned for debugging so that they can duplicate test conditions. It is also useful for tests that need to be repeated after defect repair. Scientists expect experiments to be repeatable by others, and testers should expect the same!

Principle 9. Testing should be planned.
Test plans should be developed for each level of testing, and objectives for each level should be described in the associated plan. Plans, with their precisely specified objectives, are necessary to ensure that adequate time and resources are allocated for testing tasks, and that testing can be monitored and managed. Test planning must be coordinated with project planning. The test manager and project manager must work together to coordinate activities. Testers cannot plan to test a component on a given date unless the developers have it available on that date. A test plan template must be available to the test manager to guide development of the plan according to organizational policies and standards.

Principle 10. Testing activities should be integrated into the software life cycle.
It is no longer feasible to postpone testing activities until after the code has been written. Test planning activities should be integrated into the software life cycle starting as early as in the requirements analysis phase, and continue on throughout the software life cycle in parallel with development activities. These activities can continue on until the software is delivered to the users.

Principle 11. Testing is a creative and challenging task

Difficulties and challenges for the tester

- A tester needs to have comprehensive knowledge of the software engineering discipline.
- A tester needs to have knowledge from both experience and education as to how software is specified, designed, and developed.
- A tester needs to be able to manage many details.
- A tester needs to have knowledge of fault types and where faults of a certain type might occur in code constructs.
- A tester needs to reason like a scientist and propose hypotheses that relate to presence of specific types of defects.
- A tester needs to have a good grasp of the problem domain of the software that he/she is testing. Familiarly with a domain may come from educational, training, and work-related experiences.
- A tester needs to create and document test cases. To design the test cases the tester must select inputs often from a very wide domain. Those selected should have the highest probability of revealing a defect (Principle 2). Familiarly with the domain is essential.
- A tester needs to design and record test procedures for running the tests.
- A tester needs to plan for testing and allocate the proper resources.
- A tester needs to execute the tests and is responsible for recording results.

- A tester needs to analyze test results and decide on success or failure for a test. This involves understanding and keeping track of an enormous amount of detailed information. A tester may also be required to collect and analyze test-related measurements.
- A tester needs to learn to use tools and keep abreast of the newest test tool advances.
- A tester needs to work and cooperate with requirements engineers, designers, and developers, and often must establish a working relationship with clients and users.
- A tester needs to be educated and trained in this specialized area and often will be required to update his/her knowledge on a regular basis due to changing technologies.

1.6 THE TESTER'S ROLE IN A SOFTWARE DEVELOPMENT ORGANIZATION

Testing is sometimes erroneously viewed as a destructive activity. The tester's job is to,

- reveal defects,
- find weak points,
- inconsistent behavior, and
- circumstances where the software does not work as expected.

It is difficult for developers to effectively test their own code (Principles 3 and 8). Developers view their own code as their creation, their "baby," and they think that nothing could possibly be wrong with it!

A tester requires extensive programming experience in order to understand,

- how code is constructed, and
- where, and what kind of, defects are likely to occur.

Testers also need to work along side with

Requirements Engineers	→	To ensure that requirements are testable, and to plan for system & acceptance test
Designers	→	To plan for integration and unit test.
Project Managers	→	To develop reasonable test plans,
Upper Management	→	To provide input for the development and maintenance of organizational testing standards, polices, and goals.

software quality assurance staff and software engineering process group members.

In view of these requirements for multiple working relationships, communication and team working skills are necessary for a successful career as a tester.

Developers, analysts, and marketing staff need to realize that testers add value to a software product in that they detect defects and evaluate quality as early as possible in the software life cycle. This ensures that developers release code with few or no defects, and that marketers can deliver software that satisfies the customers' requirements, and is reliable, usable, and correct.

1.7 ORIGIN OF DEFECTS

Defects as shown in Figure 3.1

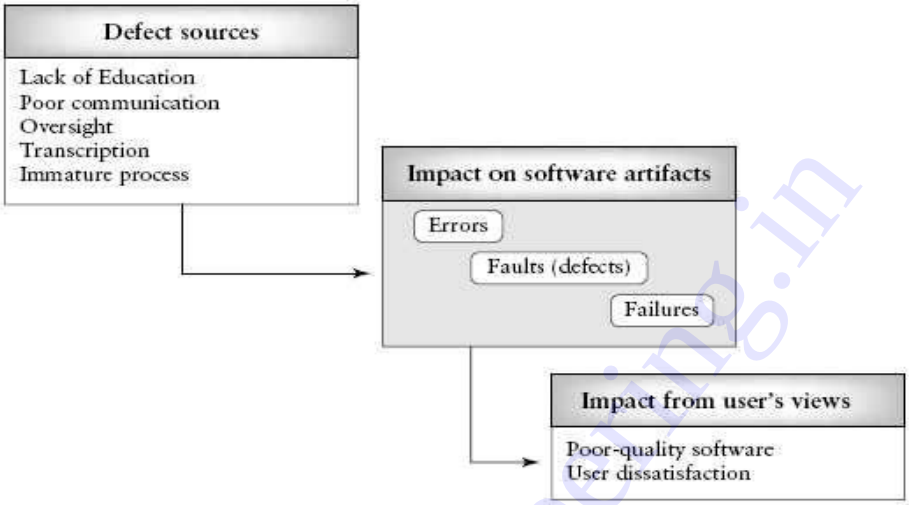


FIG. 3.1
Origins of defects.

Types of defects.

- Defect from product specification → The product developed varies in the product specification
- Variance from customer/user expectation → The user wanted that is not in the built product.

The three categories of defects are, Wrong, Missing and Extra.

Defect Sources.

1. Education → not knowing operator precedence rule in a languate
2. Communication → engineer 1 does not say about "error checking code is not there" to engineer 2.
3. Oversight → Omission of initialization statement
4. Transcription → Misspelling a variable name at the time of entering a code
5. Process → A process did not permit to use sufficient time for a detailed specification.

In order to identify these types of defects before involving the software in an operation, test case must be developed. To do so, the tester has to set hypotheses. Generally hypothesis is used to,

- Design the test case
- Design the test procedure
- Combine the test sets
- Choose the levels of testing
- To check the test results.

Fault Model.

It is a link between the error occurred and the fault/defect in the software. Physical defects normally happened due to, manufacturing errors, component wear out and environmental errors.

Use of fault model

- ✓ To prepare a fault list
- ✓ To choose a fault
- ✓ Test input prepared
- ✓ Test effectiveness is measured

1.8 DEFECT CLASSES, REPOSITORY AND TEST DESIGN

Requirements and Specification Defects

The beginning of the software life cycle is critical for ensuring high quality in the software being developed. Defects injected in early phases can persist and be very difficult to remove in later phases.

1 . Functional Description Defects
The overall description of what the product does, and how it should behave (inputs/outputs), is incorrect, ambiguous, and/or incomplete.

2 . Feature Defects
Features may be described as distinguishing characteristics of a software component or system. Features refer to functional aspects of the software that map to functional requirements as described by the users and clients.

3 . Feature Interaction Defects
These are due to an incorrect description of how the features should interact. For example, suppose one feature of a software system supports adding a new customer to a customer database. This feature interacts with another feature that categorizes the new customer. The classification feature impacts on where the storage algorithm places the new customer in the database, and also affects another feature that periodically supports sending advertising information to customers in a specific category. When testing we certainly want to focus on the interactions between these features.

4 . Interface Description Defects
These are defects that occur in the description of how the target software is to interface with external software, hardware, and users.

Design Defects

Design defects occur when system components, interactions between system components, interactions between the components and outside software/hardware, or users are incorrectly designed.

1 . Algorithmic and Processing Defects

These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. For example, the pseudo code may contain a calculation that is incorrectly specified, or the processing steps in the algorithm written in the pseudo code language may not be in the correct order. In the latter case a step may be missing or a step may be duplicated. Another example of a defect in this subclass is the omission of error condition checks such as division by zero. In the case of algorithm reuse, a designer may have selected an inappropriate algorithm for this problem

2 . Control, Logic, and Sequence Defects

Control defects occur when logic flow in the pseudo code is not correct. For example, branching to soon, branching to late, or use of an incorrect branching condition. Other examples in this subclass are unreachable pseudo code elements, improper nesting, improper procedure or function calls. Logic defects usually relate to incorrect use of logic operators, such as less than (<), greater than (>), etc. These may be used incorrectly in a Boolean expression controlling a branching instruction.

3 . Data Defects

These are associated with incorrect design of data structures. For example, a record may be lacking a field, an incorrect type is assigned to a variable or a field in a record, an array may not have the proper number of elements assigned, or storage space may be allocated incorrectly. Solution : data dictionary

4 . Module Interface Description Defects

These are defects derived from, for example, using incorrect, and/or inconsistent parameter types, an incorrect number of parameters, or an incorrect ordering of parameters.

5 . Functional Description Defects

The defects in this category include incorrect, missing, and/or unclear design elements. For example, the design may not properly describe the correct functionality of a module. These defects are best detected during a design review.

6 . External Interface Description Defects

These are derived from incorrect design descriptions for interfaces with COTS components, external software systems, databases, and hardware devices (e.g., I/O devices). Other examples are user interface description defects where there are missing or improper commands, improper sequences of commands, lack of proper messages, and/or lack of feedback messages for the user.

Coding Defects

Coding defects are derived from errors in implementing the code. Coding defects classes are closely related to design defect classes especially if pseudo code has been used for detailed design. Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designers. Others may have transcription or omission origins.

1 . Algorithmic and Processing Defects

unchecked overflow and underflow conditions, comparing inappropriate data types, converting one data type to another, incorrect ordering of arithmetic operators (perhaps due to misunderstanding of the precedence of operators), misuse or omission of parentheses, precision loss, and incorrect use of signs.

2 . Control, Logic and Sequence Defects

On the coding level these would include incorrect expression of case statements, incorrect iteration of loops (loop boundary problems), and missing paths.

3 . Typographical Defects

These are principally syntax errors, for example, incorrect spelling of a variable name, that are usually detected by a compiler, self-reviews, or peer reviews.

4 . I n i t i a l i z a t i o n Defects

These occur when initialization statements are omitted or are incorrect. This may occur because of misunderstandings or lack of communication between programmers, and/or programmers and designers, carelessness, or misunderstanding of the programming environment.

5 . Data-Flow Defects

There are certain reasonable operational sequences that data should flow through. For example, a variable should be initialized, before it is used in a calculation or a condition. It should not be initialized twice before there is an intermediate use. A variable should not be disregarded before it is used.

6 . Data Defects

These are indicated by incorrect implementation of data structures. For example, the programmer may omit a field in a record, an incorrect type or access is assigned to a file, an array may not be allocated the proper number of elements. Other data defects include flags, indices, and constants set incorrectly.

7 . Module Interface Defects

As in the case of module design elements, interface defects in the code may be due to using incorrect or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters.

8 . Code Documentation Defects

When the code documentation does not reflect what the program actually does, or is incomplete or ambiguous, this is called a code documentation defect. Incomplete, unclear, incorrect, and out-of-date code documentation affects testing efforts.

9 . External Hardware, Software Interfaces Defects

These defects arise from problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling, data exchanges with hardware, protocols, formats, interfaces with build files, and timing sequences

Testing Defects

1 . Test Harness Defects

In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code. Test harnesses are subject to the same types of code and design defects that can be found in all other types of software.

2 . Test Case Design and Test Procedure Defects

These would encompass incorrect, incomplete, missing, inappropriate test cases, and test procedures. Sometimes the defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

1.9 DEFECT EXAMPLE

Design Description for Program calculate_coin_values

Program calculate_coin_values

```
number_of_coins is integer
total_coin_value is integer
number_of_rupees is integer
number_of_paise 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_rupees = total_coin_value/100
        number_of_paise = total_coin_value - 100 * number_of_rupees
        output (number_of_rupees, number_of_paise)
end
```

```
main ()
{
  int total_coin_value;
  int number_of_notes = 0;
  int number_of_rupees = 0;
  int number_of-paise = 0;
  int coin_values = {1,5,10,25,25,100};
{
        int i = 1;
        while ( i < 6)
        {
                printf("input number of notes\n");
                scanf ("%d", number_of_notes);
                total_coin_value = total_coin_value + (number_of_notes * coin_value{i});
        }
i = i + 1;
number_of_rupees = total_coin_value/100;
number_of_paise = total_coin_value - (100 * number_of_rupees);
printf("%d\n", number_of_rupees);
printf("%d\n", number_of-paise);
}
```

Functional Description Defects

The functional description defects arise because the functional description is ambiguous and incomplete. It does not state that the input, number_of_coins, and the output, number_of_rupees and number_of_paise should all have values of zero or greater. The number_of_coins cannot be negative, and the values in dollars and cents cannot be negative in the real-world domain. As a consequence of these ambiguities and specification incompleteness, a checking routine may be omitted from the design, allowing the final program to accept negative values for the input number_of_coins for each of the denominations, and consequently it may calculate an invalid value for the results.

Solution : Formal declaration of precondition and post condition

A precondition is a condition that must be true in order for a software component to operate properly.
Example: number_of_coins >= 0
A postcondition is a condition that must be true when a software component completes its operation properly.
Example : number_of_rupees, number_of_paises >= 0.

Design 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)

Algorithmic, and processing defects. These arise from the lack of error checks for incorrect and/or invalid inputs, lack of a path where users can correct erroneous inputs, lack of a path for recovery from input errors.

Data defects. This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read 1,5,10,25,50,100.
Following sample code shows the code for the coin problem in a "C-like" programming language. Without effective reviews the specification and design defects could propagate to the code.

Coding defects

Control, logic, and sequence defects. These include the loop variable increment step which is out of the scope of the loop. Note that incorrect loop condition ($i < 6$) is carried over from design and should be counted as a design defect.

Algorithmic and processing defects. The division operator may cause problems if negative values are divided, although this problem could be eliminated with an input check.

Data Flow defects. The variable total_coin_value is not initialized. It is used before it is defined.

Data Defects. The error in initializing the array coin_values is carried over from design and should be counted as a design defect.

External Hardware, Software Interface Defects. The call to the external function "scanf" is incorrect. The address of the variable must be provided (&number_of_coins).

Code Documentation Defects. The documentation that accompanies this code is incomplete and ambiguous. It reflects the deficiencies in the external interface description and other defects that occurred during specification and design. Vital information is missing for anyone who will need to repair, maintain or reuse this code.

1.10 DEVELOPER/TESTER SUPPORT FOR DEVELOPING A DEFECT REPOSITORY

It is important if you are a member of a test organization to illustrate to management and your colleagues the benefits of developing a defect repository to store defect information. As software engineers and test specialists we should follow the examples of engineers in other disciplines who have realized the usefulness of defect data.

A requirement for repository development should be a part of testing and/or debugging policy statements. You begin with development of a defect classification scheme and then initiate the collection defect data from organizational projects. Forms and templates will need to be designed to collect the data. You will need to be conscientious about recording each defect after testing, and also recording the frequency of occurrence for each of the defect types. Defect monitoring should continue for each on-going project. The distribution of defects will change as you make changes in your processes. The defect data can support debugging activities as well.

2.1 INTRODUCTION TO TESTING DESIGN STRATEGIES

Generally software testing refers a set of activities which are well planned in advance and also conducted systematically.

The testing strategy should be flexible to the people like (i) customer and (ii) developer. The testing strategy is developed by (i) Software developers, (ii) System engineers and (iii) Testing specialists.

2.2 THE SMARTER TESTER

Before delivering the software to customer, it must be subject to test. The objectives of the test are,

- Find out the defects
- Evaluate software performance, usability and readability.

Based on these goals, the tests should be well designed. This is done by software testers. They have to choose a finite number of test cases from a large execution domain. The testing may be done based on,

- Budget
- Time constraints.

If the testing is not planned correctly and expectations are unrealistic then testers may face problems from management and marketing side.

Smarter Tester

- Provides plan for testing
- Selecting the test cases
- Monitoring the process to ensure the resources and time allotted for the work are used effectively.

To do this, the smarter tester requires,

- Proper education and training
- Ability to enlist the management support

Novice Tester

- Test a module or component by employing all set of inputs
- Check all the software structures

Goal of smart tester.

- Understand the functionality
- Know the input / output domain
- Understand the environment use for the code being tester

2.3 TEST CASE DESIGN STRATEGIES

An effective test case should find the maximum number of defects. The test cases are important to an organization. When the test cases are effective,

- It is possible to find maximum possibility of errors
- The resources of an organization is effectively used
- High probability for test reuse.
- Possible to give the closer adherence to testing and project schedules and budgets
- Higher quality of software can be delivered.

There are two basic strategies that can be used to design test cases. These are called the black box (sometimes called functional or specification) and white box (sometimes called clear or glassbox) test strategies. The approaches are summarized in the following table.

Test Strategy	Tester's View	Knowledge Source	Methods
Black box	Process flow not known	Requirements document Specifications Domain knowledge Defect analysis data	Equivalence class partitioning Boundary value analysis State transition testing Cause and effect graphing Error guessing
White box	Process flow known	High-level design Detailed design Control flow graphs Cyclomatic complexity	Statement testing Branch testing Path testing Data flow testing Mutation testing Loop testing

2.4 USING BLACK BOX APPROACH TO TEST CASE DESIGN

Black box testing involves looking at the specifications and does not require examining the code of a program. The test engineer engaged in black box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software.

Example : Lock and key

We do not know how the levers in the lock work, but we only know the set of inputs (no. of keys, how to use the key, direction of turn of each key) and the expected outcome (lock and unlock). Some of the functionalities that you need to know to use the lock are given below.

Black box testing thus requires a functional knowledge of the product to be tested. It does not mandate the knowledge of the internal logic of the system nor does it mandate the knowledge of the programming language used.

Functionality	What you need to know to use
Features of a lock	Made of metal, has a hole provision to lock, facility to insert the key, hole ability to turn the key
Features of a key	Made of metal, created to fit into a particular lock's keyhole
Actions performed	Key inserted and turned clock or anti clockwise to lock or unlock
States	Locked, Unlocked
Input	Key turned clock and anticlockwise
Expected output	Locking, Unlocking

Why black box testing.

- Done based on requirements
- Addresses the stated requirements as well as implied requirements
- Encompasses the end user perspective
- Handles valid and invalid inputs

2.4.1 RANDOM TESTING

Random testing is accomplished by randomly selecting the test cases. This approach has the advantage of being fast and it also eliminates biases of the testers. Here the inputs are chosen randomly from the domain.

Eg. Valid input domain for a module → all possible integers from 1 to 100. Now the tester has to choose the input values from the given domain, and let it be 2, 9, 83.

Possible questions arises.

- It is enough to prove that the module meets its specification at the time of tests conducted?
- Any additional values required making effective use of the resources?
- Any other input values find more number of defects than these selected values?
- Any extra value like floating point values, negative values or the value greater than 100 will find more number of defects.
- Advantage → Saves time and effort
- Disadvantage → way of testing creates only minimum possibility of getting test data.

2.4.2 REQUIREMENTS BASED TESTING

Requirement based testing deals with validating the requirements given in the software requirement specifications (SRC). Explicit requirements are stated and documented as part of the requirement specification. Implicit requirements are those that are documented but assumed to be incorporated in the system. A requirements specification for the lock and key example can be documented as given in the following table.

Requirements are tracked by a Requirements Traceability Matrix (RTM). An RTM traces all the requirements from their genesis through design, development and testing. The mapping between requirements and the testing are listed in this example.

S.No.	Requirments ID	Description	Priority
1	BR-01	Insert key number 123 and turn clockwise, facilitate locking	H
2	BR-02	Insert key number 123 and turn anti-clockwise, facilitate unlocking	H
3	BR-03	Only key number 123 can be used to lock and unlock	H
4	BR-04	No other object can be used to lock	M
5	BR-05	No other object can be used to unlock	M
6	BR-06	The lock must not open even when it is hit with a heavy object	M
7	BR-07	Lock and key made of metal and weigh 150 gms.	L
8	BR-08	Lock and Unlock directions should be changeable for usability of left-handers	L

Requirments ID	Test Condition	Test case IDs
BR-01	Use key 123	Lock-001
BR-02	Use key 123	Lock-002
BR-03	Use key 123 to lock Use key 123 to unlock	Lock-003 Lock-004
BR-04	Use key 456 Use hairpin Use screwdriver	Lock-005 Lock-006 Lock-007
BR-05	Use key 456 Use hairpin Use screwdriver	Lock-008 Lock-009 Lock-010
BR-06	Use stone to break the lock	Lock-011
BR-07	Use weighing machine	Lock-012
BR-08	Lock and Unlock directions should be changeable for usability of left-handers	-

Once the test case creating is completed, the RTM helps in identifying the relationship between the requirements and test cases. The following combinations are possible

- One to one → For each requirement there is one test case (eg. BR-01)
- One to many → For each requirement there are many test cases (eg. BR-03)
- Many to one → A set of requirements can be tested by one test case (Nil)
- Many to many → Many requirements can be tested by many test cases (Nil)
- One to none → The set of requirements can have no test cases (eg. BR-08)

Req. ID	Test cases	Total test cases	Test cases passed	Test cases failed	% Pass	No. of defects
BR-01	Lock-01	1	1	0	100	1
BR-02	Lock-02	1	1	0	100	1
BR-03	Lock-03,04	2	1	1	50	3
BR-04	Lock-05,06,07	3	2	1	67	5
BR-05	Lock-08,09,10	3	3	0	100	1
BR-06	Lock-11	1	1	0	100	1
BR-07	Lock-12	1	1	0	100	0
BR-08	-	0	0	0	0	1
Total	8	12	10	2	83	12

Role of RTM

1. RTM gives a tool to know the testing status of every requirement without missing any requirement
2. It allows the testers to prioritize the test cases execution to find the defects.
3. Also it is used to find whether there are adequate test cases for high priority requirements.
4. The collection of test cases which says the requirements can be viewed from the RTM

To illustrate the metric analysis, let us assume the test execution data as given in the above table. From the above table, the following observations can be made with respect to the requirements.

- 83% passed test cases correspond to 71% of requirements being met (5 out of 7).
- From the failed test cases, outstanding defects affect 29% of the requirements.
- A high priority requirement, BR-03, which has failed. There are 3 corresponding defects that need to be looked into and test case Lock-04 need to be executed again.
- A medium-priority requirement, BR-04, has failed. Test case Lock-06 has to be re-executed after the defects are fixed.
- The requirement BR-08 is not met, however, this can be ignored for the release, even though there is a defect, due to the low priority nature of this requirement.

The metrics discussed can be expressed in graph also.

2.4.3 POSITIVE AND NEGATIVE TESTING

Positive testing tries to prove that a given products does what is is supposed to do. When a test case verifies the requirements of the product with a set of expected output, it is called positive test case.

The purpose of positive testing is to prove that the product works as per specifications and expectations. A product delivering an error when it is expected to given an error, is also a part of positive testing.

Eg. Row 1 → When the lock is in an unlocked state and we use key 123 and turn it clockwise, the expected outcome is to get it locked. During test execution, if the test results in locking, then the test is passed. This is an example of “positive test condition” for positive testing.

Req. ID	Input 1	Input 2	Current State	Expected output
BR-01	Key 123	Clockwise	Unlocked	Locked
BR-01	Key 123	Clockwise	Locked	Unlocked
BR-02	Key 123	Anticlockwise	Unlocked	Locked
BR-02	Key 123	Anticlockwise	Locked	Unlocked
BR-04	Hairpin	Clockwise	Locked	No change

Eg. Row 5 → Using a hairpin and turn it clockwise should not cause a change in state or any damage. On test execution, if there are no changes, then the positive test case is passed. This is an example of a "negative test condition" for positive 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 and break the system.

S.No.	Input 1	Input 2	Current State	Expected output
1	Key 456	Clockwise	Locked	Locked
2	Key 456	Anticlockwise	Unlocked	Locked
3	Wire	Anticlockwise	Unlocked	Unlocked
4	Stone hit	-	Locked	Locked

In this above table note that, there is no requirement ID. This is because negative testing focuses on test conditions that lie outside the specification.

The difference between positive testing and negative testing is in their coverage. For positive testing if all documented requirements and test conditions are covered, then coverage can be considered to be 100%. In contrast, there is no end to negative testing, and 100% coverage is impractical in negative testing.

2.4.4 BOUNDARY VALUE ANALYSIS

The two major sources of defects in a software product are conditions and boundaries. By boundaries, we mean "limits" of values of a various variables. This method is useful in arriving at tests that are effective in catching defects that happens at boundaries. Consider a hypothetical store that sells certain commodities and offers different pricing for people buying in different quantities.

Number of units bought	Price per unit
1 -10	Rs. 50
11 – 20	Rs. 45
21 – 30	Rs. 42
More than 30	Rs. 40

The question is what test data is likely to reveal the most number of defects in the program? It has been found that most defects in situations such as this happen around the boundaries – for

example, when buying 9, 10, 11, 19, 20, 21, 29, 30, 31 etc. Some possible reasons for this phenomenon are as follows,

- Programmers' tentativeness in using the comparison operators (eg. <= or < ?)
- Confusing caused by the availability of multiple ways to implement loops and conditions checking (eg. for, while or repeat ?)
- The requirements themselves may not be clearly understood, especially around the boundaries.

2.4.5 DECISION TABLES

A decision table denotes,

1. List of decision variables

The variables related to the decision are given as the columns of a decision table. However, when more number of decision variables or less number of distinct combinations of variable, these may be given as rows.

2. Conditions set for every decision variable
All the conditions assumed for every decision variable is continuously given one after the other.

3. Actions to be done in each combination of the conditions.

The actions are listed as last column of the decision tables. If the value of decision variable does not affect the outcome of decision then entries are marked as "-". These are said to be "don't cares". It minimizes the total number of tests to be done.

Eg: Decision table for finding Fee structure.

Status	Status of student		Whether to join I year	Whether to join LE	Fee details
	Merit	Management			
ECE	Yes	No	Yes	No	Rs.20000
	No	Yes	No	Yes	Rs.40000
IT	Yes	No	No	Yes	Rs.10000
	No	Yes	Yes	No	Rs.15000
CSE	Yes	No	Yes	No	Rs.18000
M.E. (CSE)	-	-	Yes	No	Rs.25000
CE	-	-	No	Yes	Rs.21000

Steps to follow in a decision table creation.
1. Decision variable are first found
2. Find the values of every decision variable
e. Create the combination of values of decision variables.
4. Identify the cases when values assumed by a variable are immaterial for a given combination of other input variables.

5. It must to print the expected result for every combination at decision variable values
6. By putting the decision variable in the column wise and at last it should be the action for the combination of decision variable in that row.

The main use of this method is, as invaluable tools for designing black box tests to check the behavior of product under various logical conditions of input variables.

2.4.6 EQUIVALANCE CLASS PARTITIONING

If a tester is viewing the software-under-test as a black box with well defined inputs and outputs, a good approach to selecting test inputs is to use a method called equivalence class partitioning.

Equivalence partitioning is a software testing technique that involves identifying a small set of representative input values that produce as many different output conditions as possible. This reduces the number of permutations and combinations of input, output values used for testing. The set of input values that generate one single expected output is called ***partition***. When the behavior of the software is the same for a set of values then the set is termed as an ***equivalence class or a partition***.

Using equivalence class partitioning a test value in a particular class is equivalent to a test value of any other member of that class. Therefore, if one test case in a particular equivalence class reveals a defect, all the other test cases based on that class would be expected to reveal the same defect. We can also say that if a test case in a given equivalence class did not detect a particular type of defect, then no other test case based on that class would detect the defect

Testing by this technique involves,

- identifying all partitions for the complete set of input, output values for a product
- Picking up one member values from each partition for testing to maximize complete coverage.

Advantages:

1. It eliminates the need for exhaustive testing, which is not feasible.
2. It guides a tester in selecting a subset of test inputs with a high probability of detecting a defect.
3. It allows a tester to cover a larger domain of inputs/outputs with a smaller subset selected from an equivalence class.

Important points related to equivalence class partitioning

1. The tester must consider both valid and invalid equivalence classes. Invalid classes represent erroneous or unexpected inputs.
2. Equivalence classes may also be selected for output conditions.
3. Given the same set of conditions, individual testers may make different choices of equivalence classes. As a tester gains experience he is more able to select equivalence classes with confidence.
4. A tester should also realize that for some software problem domains defining equivalence classes is inherently difficult, for example, software that needs to utilize the tax code.

Example

LIC's base premium – Rs. 500, based on age group, an additional premium has to be paid. Age <35 → Rs.2500, Age 35 – 59 → Rs.5000, Age 60+ → Rs.10000

The equivalence partitions that are based on age given below,

- ✓ Below 35 years of age - valid input
- ✓ 35 – 59 - valid input
- ✓ >= 60 - valid input
- ✓ Negative age - invalid input
- ✓ Age as zero (0) - invalid input
- ✓ Age as any 3-digit - valid input

The test case for this example based on equivalence partition is given below.

SNo.	Equivalance partition	Type of input	Test data	Expected results
1	Below 35 years of age	Valid	26, 12	Rs.3000
2	35 – 59	Valid	37	Rs. 5500
3	>= 60	Valid	65, 90	Rs.10500
4	Negative age	Invalid	-23	Warning message
5	Age as 0	Invalid	0	Warning message

Few other ways to identify equivalence classes using ranges of values are ***Prime numbers, Composite numbers, Numbers with decimal points***.

Equivalence partitioning is useful to minimize the number of test cases when the input data can be divided into distinct sets, where the behavior or outcome of the product within each member of the set is the same.

2.4.7 STATE BASED TESTING

State or graph based testing is very useful in situations where,

1. The product under test is a language processor (eg. Compiler)
2. Workflow modeling
3. Dataflow modeling

Example.

Consider an application that validates a number,

- A number starts with an optional sign
- The optional sign can be followed by any number of digits
- The digits can be optionally followed by a decimal point
- If decimal point, then there should be two digits after it
- Any number, should be terminated by a blank

The above rules can be represented in a state transition diagram as below.

The state transition diagram can be converted to a 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 table can be used to derive test cases to test valid and invalid numbers. Some of them are,

1. Start from the start state (state 1)
2. Choose a path that leads to the next state (eg. +/-/digit, state 1 to 2)
3. If you encounter an invalid input in a given state (alphabetic character in state 2), generate an error condition test case
4. Repeat the process till you reach the final state (state 6)

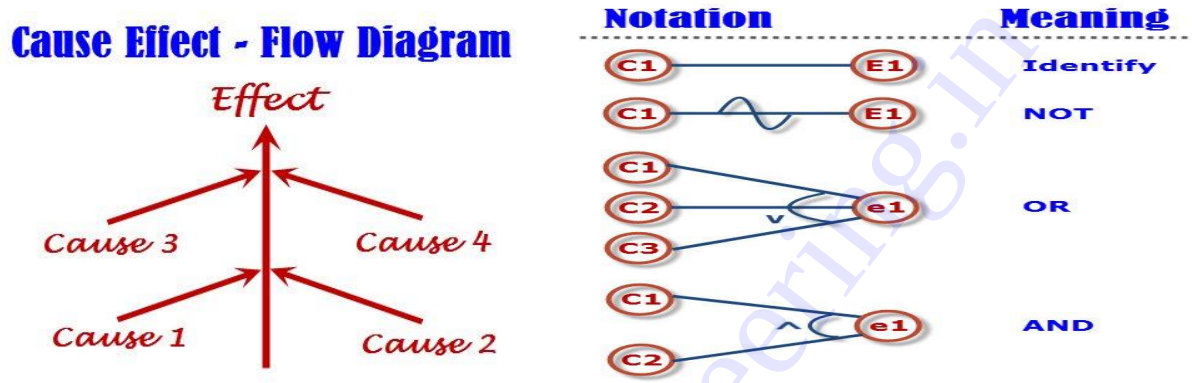
2.4.8 CAUSE EFFECT GRAPHING

A major weakness with equivalence class partitioning is that it does not allow testers to combine conditions. Combinations can be covered in some cases by test cases generated from the classes. ***Cause-and-effect graphing is a technique that can be used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification***.

Developing the graph, especially for a complex module with many combinations of inputs, is difficult and time consuming. The graph must be converted to a decision table that the tester uses to develop test cases. The steps in developing test cases with a cause-and-effect graph are,

1. The tester must decompose the specification of a complex software component into lower-level units.
2. For each specification unit, the tester needs to identify causes and their effects. ***A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation***. The logical relationships between the causes and effects should be determined

3. From the cause-and-effect information, a Boolean cause-and-effect graph is created. Nodes in the graph are causes and effects**.** ***Causes*** are placed on the left side of the graph and effects on the right.
4. 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.
5. The graph is then converted to a decision table.
6. The columns in the decision table are transformed into test cases.



Symbols used in Cause-Effect graph

Example: Suppose we have a specification for a module that allows a user to perform a search for a character in an existing string. The specification states that the user must input the length of the string and the character to search for. If the string length is out-of-range an error message will appear. If the character appears in the string, its position will be reported. If the character is not in the string the message "not found" will be output.

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.

Based on the causes, effects, and their relationships, a cause-and-effect graph to represent this information is shown in following figure.

The next step is to develop a decision table. The decision table reflects the rules and the graph and shows the effects for all possible combinations of causes.

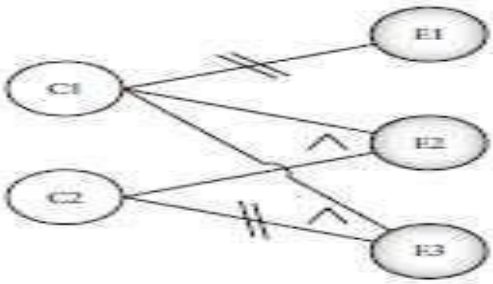


FIG. 4.5
Cause-and-effect graph for the character search example.

A decision table will have a row for each cause and each effect. The entries are a reflection of the rules and the entities in the cause and effect graph. 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 following table where C1, C2, C3 represent the causes, E1, E2, E3 the effects, and columns T1, T2, T3 the test cases.

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

Advantages

1. Development of the rules and the graph from the specification allows a thorough inspection of the specification. Any omissions, inaccuracies, or inconsistencies are likely to be detected.
2. Other advantages come from exercising combinations of test data that may not be considered using other black box testing techniques.

The major problem is developing a graph and decision table when there are many causes and effects to consider. ***A possible solution*** to this is to decompose a complex specification into lower-level, simpler components and develop cause-and-effect graphs and decision tables for these.

2.4.9 ERROR GUESSING

Designing test cases using the error guessing approach is based on the ***tester's/developer's past experience*** with code similar to the code-under test, and ***their intuition*** as to where defects may lurk in the code. Code similarities may extend to the structure of the code, its domain, the design approach used, its complexity, and other factors. The tester/developer is sometimes able to make an ***educated "guess"*** as to which types of defects may be present and design test cases to reveal them.

Some examples of obvious types of defects to test for are cases where there is a possible **division by zero**, where there are a **number of pointers** that are manipulated, or **conditions** around array boundaries.

Error guessing is an ad hoc approach to test design in most cases. However, if defect data for similar code or past releases of the code has been carefully recorded, the defect types classified, and failure symptoms due to the defects carefully noted, this approach can have some structure and value.

2.4.10 COMPATABILITY TESTING

The integrity of the testing generally depends on the infrastructure for delivering functionality. When infrastructure parameters are changes, the products is expected to still behave correctly and produce the desired or expected results. As a result, the compatibility testing ensures the working of the product with different infrastructure components.

Testing done to ensure that the product features work consistently with different infrastructure components is called compatibility testing.

The parameters that generally affect the compatibility of the product are,

- Processor and the number of processor (p3,p4, xenon etc)
- Architecture and characteristics of the machine (32 bit, 64 bit…)
- Resource availability on the machine (RAM, disk space, network card..)
- Equipment that the product is expected to work (printers, modem, router….)
- Operating system and its services
- Backend components such as database servers (oracle, Sybase…)
- Any software used to generate product binaries (compiler, linker,…)

In order to arrive at practical combinations of the parameters to be tested, a compatibility matrix is created. It has as its columns various parameters the combinations of which have to be tested as shown below.

Common techniques that are used to perform compatibility testing are,

1. Horizontal combination
2. Intelligent sampling

The compatibility testing of a product involving parts of itself can be further classified into two types,

- Backward compatibility testing: It is important for the customers that the objects, object properties, schema, rules, reports etc. that are created with an older version of the product continue to work with current version also. The testing ensures this is called backward compatibility testing.
- Forward compatibility testing: There exist some provisions for the product to work with later versions of the product and other infrastructure components, keeping future requirements in mind. Eg. IP network protocol version 6 used 128 bit addressing…)

2.4.11 DOCUMENTATION TESTING

User documentation covers all the manuals, user guides, installation guides, setup guides, read me file, software release notes and online help that are provided along with the software to help the end user to understand the software system.

Objectives of user documentation testing

- 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.

User documentation testing focuses on ensuring what is in the document exactly matches the product behavior, by sitting in front of the system and verifying screen by screen, transaction by transaction and report by report.

Since these documents are the first interactions the users have with the product, they tend to create lasting impressions. A badly written installation document can put off a user and bias him or her against the product, even if the product offers rich functionality.

Advantages.

1. Aids in highlighting problems over looked during reviews
2. High quality user documentation ensures consistency of documentation and product, thus minimizing possible defects reported by customers.
3. Results in less difficult support calls.
4. New programmers and testers who join a project group can use the documentation to learn the external functionality of the product.
5. Customers need less training and can proceed more quickly to advanced training and product usage.

2.4.12 DOMAIN TESTING

Domain testing is testing the product, purely based on domain knowledge and expertise in the domain of application.

This testing approach requires critical understanding of the day-to-day business activities for which the software is written. The test engineers performing this type of testing are selected because they have in-depth knowledge of the business domain. This reduces the effort and time required for training the testers in domain testing and also increases the effectiveness of domain testing.

Example : Banking software; Training the bank official with the testing knowledge is better than training the tester with banking knowledge.

Domain knowledge is the ability to design and execute test cases that relate to the people who will buy and use the software. It helps in understanding problems they are trying to solve and the ways in which they are using the software to solve them. It is also characterized by how well an individual test engineer understands the operation of the system and the business processes that system is supposes to support.

Domain testing involves testing the product, not by going through the logic built into the product. The business flow determines the steps, not the software under test. This is also called as "***business vertical testing***".

2.5 USING WHITE BOX TESTING APPROACH TO TEST DESIGN

White box testing is a way of testing the external functionality of the code by examining and testing the program code. This is also known as clear box, glass box or open box testing.

The defects in which the white box testing is useful are,

- Incorrect translation of requirements and design into program code
- Programming errors

2.6 TEST ADEQUACY CRITERIA

The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements such as statements and branches. Testers 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. Such a framework exists in the form of test adequacy criteria. The criteria can be viewed as representing minimal standards for testing a program. The application scope of adequacy criteria also includes:

- 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 the development of quantitative objectives for testing;
- indicating to testers whether or not testing can be stopped for that program.

A program is said to be adequately tested with respect to a given criterion if all of the target structural elements have been exercised according to the selected criterion. If a test data adequacy criterion focuses on the structural properties of a program it is said to be a program-based adequacy criterion. Program-based adequacy criteria are commonly applied in white box testing. They use logic and control structures, data flow, program text, or faults as the focal point of an adequacy evaluation.

An adequacy criterion that focuses on statement/branch properties is expressed as the following:

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.

The concept of test data adequacy criteria, and the requirement that certain features or properties of the code are to be exercised by test cases, leads to an approach called "coverage analysis,"

which in practice is used to set testing goals and to develop and evaluate test data. When coverage related testing goal is expressed as a percent, it is often called the "degree of coverage."

The planned degree of coverage may be less than 100% possibly due to the following:

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

2.7 STATIC TESTING VS STRUCTURAL TESTING

2.7.1 STATIC TESTING

Static testing is a type of testing which requires only the source code of the product, not the binaries or executables. It does not involve executing the programs on computers, but involves select people going through the code to find out whether,

- Code works according to requirements
- Code developed in accordance with design
- Code handles errors properly
- Code for any functionality has been missed out

Static testing by humans

This method relies on the principle of ***humans reading*** the program code to detect errors ***rather than computers executing*** the code to find errors.

Advantages.

- ✓ Sometimes humans can find errors that computers cannot.
- ✓ Multiple humans read and evaluate the program, more problems identified upfront than a computer could.
- ✓ Human evaluation compares the code with design or specification to ensure that it does what is intended to do.
- ✓ Human evaluation detect many problems at one go and can even try to identify the root causes of the problems.
- ✓ Human tests before code execution, saves computer resources (at the expense of human resources).
- ✓ Static testing minimizes the delay in problem identification.
- ✓ Finding defects in later stage creates immense pressure on programmers and thus causes other defects due to time constraint.

Methods for static testing

Desk checking

- ***Desk checking*** is a method to verify the portions of the code for correctness.
- Normally done manually by the author of the code, this method is done before compiling and executing the code.
- Whenever errors are found, the author applies the corrections for errors on the spot.
- This method relies completely on the author's thoroughness, diligence and skills.
- There is no process or structure that guarantees the effectiveness of this checking.
- This method is effective for correcting "obvious" coding errors but will not be effective in detecting errors that arise due to incorrect understanding of requirements.
- The defects are detected and corrected with minimum delay.

Disadvantage

- A developer is not the best person to detect problems in his own code.
- 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.

Code walkthrough

- In **code walkthrough**, a set of people look at the program code, and answers the questions.
- If the author is unable to answer some questions, he then takes those questions and finds their answers.
- Completeness is limited to the area where questions are raised by the team.

Code inspection

Code inspection (Fagan inspection) is a method, normally with high degree of formalism, focuses on detecting all faults, violations, and other side-effects. Once the code is ready for inspection after desk checking and walkthrough, an inspection meeting is arranged.

Roles of inspection team

- ***Author*** of the code, presents his perspective of what the program is intended to do.
- ***Moderator*** who is expected to formally run the inspection according to the process. They also inform the team about the date, time and venue of the meeting.
- ***Inspectors*** who actually provides, review comments for the code. They get copies of the code to be inspected along with other documents like requirements document etc.
- ***Scribe***, who takes detailed notes during the inspection meeting and circulates them to the inspection team after the meeting.

Process of inspection team

- Team assembles at the agreed time for the inspection meeting (defect logging meeting).
- Moderator takes the team sequentially through the code, asking each inspector if there are any defects in that part of the code.
- If any, the team deliberates on the defect and on agreement, the defect is classified.
    - ***Minor*** : defects that may not substantially affect a program.

- ***Major*** : need immediate attention.
- ***Systemic***: require correction at different level.
- ***Mis-execution*** : happens because of an error or slip on the part of the author.

- Scribe formally documents the defects found in the inspection meeting and the author takes care of fixing these defects.

Static analysis tools.

Several static analysis tools are available that reduces the manual work and perform analysis of the code to find out errors such as,

- Unreachable codes
- Variable declared but not used
- Mismatch in definition and assignment of values to variables
- Illegal typecasting of variables
- Use of non-portable programming constructs
- Memory allotted but no statements for freeing

For any type of human checking, it is useful to have a **code review checklist**. A checklist that covers some common issues is given below.(Refer Photostat or Text book)

2.7.2 STRUCTURAL TESTING

Structural testing takes into account the code, code structure, internal design and how they are coded. The main difference between structural testing and static testing is that in structural testing tests are actually run by the computer on the built product.

2.8.1 CODE(UNIT) FUNCTIONAL TESTING

- Initially the developer can perform certain static testing. This can be quick test that checks out any obvious mistakes. By repeating these tests for multiple values of input variables, the confidence level of the developer to go to the next level increases.
- For modules with complex logic or conditions, the developer can build a "debug version" of the product by putting intermediate print statement. This is to make sure the program is passing through the right loops and iterations the right number of times.
- Another approach is to run the product under a debugger or an Integrated Development Environment. These tools allow single stepping of instructions, setting break points at any function or instruction. Then to view the various system parameters or program variable values.

2.8.2 COVERAGE AND CONTROL FLOW GRAPHS

The application of coverage analysis is typically associated with the use of control and data flow models to represent program structural elements and data. In a most conventional programming language, program constructs can be classified as,

- ✓ Sequential flow control
- ✓ Tow-way decision statement (if then else)

- ✓ Multi-way decision statement (switch)
- ✓ Loops like while do, repeat until and for
- ✓ combinations of decisions and conditions;
- ✓ paths (node sequences in flow graphs).

2.8.3 COVERING CODE LOGIC (CODE COVERAGE TESTING)

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. The percentage is found by adopting a technique called ***instrumentation of code***. Instrumentation rebuilds the product, linking the product with a set of libraries provided by the tool vendors.

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). Graphical representations for these three primes are shown in Figure 5.1.

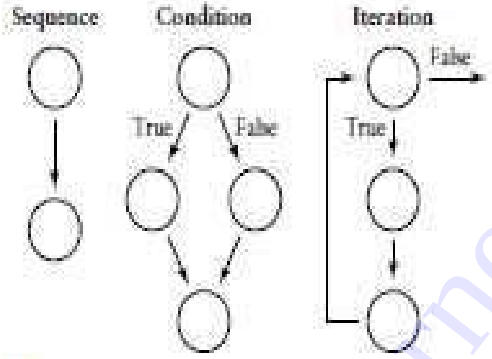


FIG. 5.1
Representation of program primes.

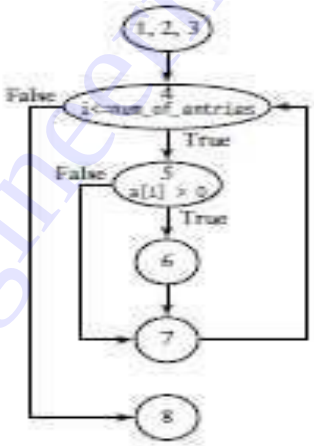


FIG. 5.3
A control flow graph representation for the code in Figure 5.2.

A flow graph representation for the code example in Figure 5.2 is found in Figure 5.3. Note that in the flow graph the nodes represent sequential statements, as well as decision and looping predicates. For simplicity, sequential statements are often omitted or combined as a block that indicates that if the first statement in the block is executed, so are all the following statements in the block. Edges in the graph represent transfer of control. The direction of the transfer depends on the outcome of the condition in the predicate (true or false).

/* pos_sum nds 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

FIG. 5.2 ***Code sample with branch and loop.***

The various types of code coverage testing are,

1. Statement coverage
2. Path coverage
3. Condition coverage
4. Function coverage

Statement Coverage

Statement coverage refers to writing test cases that execute each of the program statements. The more the code covered, the better is the testing of the functionality. Code coverage can be achieved by providing coverage to each of the above types of statements.

- For a ***section of code*** that consists of statements that are ***sequentially executed*** (without conditional branches), test cases can be designed to run through from top to bottom. Sometimes, even if we start a test case at the beginning of a section, the test case may not cover all the statements in that section. This is because, if there are asynchronous exceptions that the code encounters such as divide by zero.
- For a ***two-way decision construct***, we should cover the then and else parts of the if statement. To do this, test cases should be developed to test each then and else statement.
- In a ***multi-way decision construct***, to cover all possible switch cases, there would be multiple test cases.
- A good percentage of the defects in programs come about because of loops that do not function properly. More often, loops fail in what are called "boundary conditions". Thus test cases should be designed for,
  - Skip the loop completely – termination condition being true before state of the loop.
  - Test the loop between once and maximum number of times – check all possible normal operation
  - Try to cover the boundary – just below n, n and just above n.

The statement coverage for a program can be calculated by the formula,

Statement coverage = (total stts. exercised / total number of executable stts in program) * 100.

Based on this formula, even though we achieved a very high level of statement coverage, it does not mean that the program is defect-free for the following reasons,
The program implements wrong requirements and this wrongly implemented code is fully tested with 100 percentage coverage.

Consider an example code,

```
total = 0;
if (code == 'M')
        { stt1;stt2;stt3;stt4;stt5;stt6;stt7;}
else
        percent = value / total * 100;
```

Test with code = 'M' gives 80% code coverage. However, the probability of code not being equal to 'M' in real world data distribution is 90%. Thus the program will fail 90% of the time (divide by zero). So, even the code coverage achieves 80%, we are left with a defect that hits the users 90% of the time.

Path coverage

In ***path coverage***, we split the program into a number of paths and the program take any of the paths to its completion.

Eg. Date validation routine. For flowchart refer class notes or text.

Regardless of the number of statements in each of these paths, if we can execute these paths, then we would have covered most of the typical scenarios. Even if we have covered all the paths possible in the above example, still the program is not fully tested.

There are classes of errors which branch coverage cannot detect, such as:

```
$h = 0;
if ($x)
{
   $h = { a => 1 };
}
if ($y)
{
   print $h->{a};
}
```

100% branch coverage can be achieved by setting ($x, $y) to (1, 1) and then to (0, 0). But if we have (0, 1) then things go bang.

The purpose of path coverage is to ensure that all paths through the program are taken. In any reasonably sized program there will be an enormous number of paths through the program and so in practice the paths can be limited to those within a single subroutine, if the subroutine is not too big, or simply to two consecutive branches.

In the above example there are four paths which correspond to the truth table for $x and $y. To achieve 100% path coverage they must all be taken. Note that missing else's count as paths.

In some cases it may be impossible to achieve 100% path coverage:

```
a if $x;
b;
c if $x;
```

50% path coverage is the best you can get here. Ideally, the code coverage tool you are using will recognize this and not complain about it, but unfortunately we do not live in an ideal world. And anyway, solving this problem in the general case requires a solution to the halting problem, and I couldn't find a module on CPAN for that.

100% path coverage implies 100% branch coverage.

Condition Coverage.

In the above example even if we have covered all the paths possible, it would not mean that the program is fully tested.

Eg. Take path 'A' by giving a value < 1. Now path 'A' is covered and program has detected that the month is invalid. However, program may still not tested correctly for other condition m>12.

Another issue is, for example, when there is an OR condition, once the first part of the IF is found to be true, second part will not be evaluated at all. Similarly when there is an AND condition, if the first part is FALSE, the rest of the expression need not be evaluated at all.

Thus, the path coverage is not sufficient. Test cases for each Boolean expression have to be designed for each TRUE and FALSE condition. Eventually, this will result in more test cases and this will rise exponentially when the number of condition increases.

Condition coverage = (Total decisions exercised / Total no. of decision in program) * 100

The goal of branch coverage is to ensure that whenever a program can jump, it jumps to all possible destinations. The simplest example is a complete if statement:

```
if ($x)
{          print "a";        }
Else     {          print "b";       }
```

Full coverage is only achieved here only if $x is true on one occasion and false on another.
Achieving full branch coverage will protect against errors in which some requirements are not met in a certain branch. For example:

```
if ($x)
{          $h = { a => 1 }       }
Else     {          $h = 0;       }
print $h->{a};
```

This code will fail if $x is false

In such a simple example statement coverage is as powerful, but branch coverage should also allow for the case where the else part is missing, and in languages which support the construct, switch statements should be catered for:

```
$h = 0;
if ($x)
```

```
{
   $h = { a => 1 }
}
print $h->{a};
```

100% branch coverage implies 100% statement coverage.

Function Coverage.

To identify how many program functions are covered by test cases.

The requirements are mapped into functions during design phase. Eg. Inserting a row into the database, tax calculation in a payroll application. Test cases are written to exercise each of the different functions of the code.

Advantages.

1. Functions are easier to identify in a program and hence to write test cases.
2. Functions are much higher level abstraction than code; it is easier to achieve 100 percent of coverage.
3. The importance of a function can be prioritized based on the importance of the requirements and thus it is easier to prioritize the functions for testing.
4. Function coverage provides a natural transition to black box testing.

2.8.4 PATHS – THEIR ROLE IN WHITE–BOX BASED TEST DESIGN

CODE COMPLEXITY TESTING

At the end of all the coverage already discusses, two questions that come to mind are,

- ✓ Which of the paths are independent? If two paths are not independent, then we can minimize the number of tests.
- ✓ Is there any upper bound on the number of tests that must be run to ensure that all statements have been executed at least once?

Cyclomatic complexity is a metric than quantifies the complexity of a program.

A program is represented in the form of a flow graph. A ***flow graph*** consists of nodes and edges. To convert a standard flow chart into a flow graph to compute cyclomatic complexity, following steps are taken, (example Fig. b)

1. Identify the decision points.
2. Ensure that the decision points are simple. Eg. Fig. a
3. Combine all sequential statements into a single node (all statements executed, once started)
4. If a set of sequential statements are followed by a simple decision points, combine all of them into one node and have two edges from this node.
5. Make sure that all the edges terminate at some node.

Complexity calculations.

Consider a hypothetical program with no decision points. The flow graph will have 2 nodes and 1 edge (see fig.c). The edge is the only independent path. Hence, the cyclomatic complexity for this graph based on independent path = 1. P=0, complexity = P+1 = 1. Similarly, based on decision points, the cyclomatic complexity is also = 1 (see fig. d). E=1, N=1, Complexity = E-N+2 = 1.

Now, consider a decision point is added to this graph (see fig.f). Obviously, two paths are there (for TRUE and FALSE). The cyclomatic complexity for this graph based on independent path = 2. P=1, complexity = P+1 = 2. Similarly, based on decision points, the cyclomatic complexity is = 2 (see fig. d). E=4, N=4, Complexity = E-N+2 = 2.

Using the flow graph, an ***independent path*** can be defined as a path in the flow graph that has at least one edge that has not been traversed before in other paths. A set of independent paths that cover all the edges is a ***basis set***.

Complexity	Description
1-10	Well-written code, testability is high, maintenance cost is low
11-20	Moderately complex, testability is medium, maintenance cost is medium
21-40	Very complex, testability is low, maintenance cost is high
>40	Not testable, money / effort to maintain are not enough.

2.9 EVALUATING TEST ADEQUACY CRITERIA

As a conscientious tester one might at first reason that his testing goal should be to develop tests that can satisfy the most stringent criterion. However, one should consider that each adequacy criterion has both strengths and weaknesses. Each is effective in revealing certain types of defects. Testing conditions and the nature of the software should guide your choice of a criterion.

Weyuker presents a 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 the framework with which to evaluate these 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.

1 . Applicability Property

"For every program there exists an adequate test set". For all programs we should be able to design an adequate test set that properly tests it. If we test on all representable points, that is called an exhaustive test set. The exhaustive test set will surely be adequate since there will be no other test data that we can generate. However, exhaustive testing results in too expensive, time consuming, and impractical.

2 . Nonexhaustive Applicability Property

"For a program *P* and a test set *T*, *P* is adequately tested by the test set *T*, and *T* is not an exhaustive test set". A tester does not need an exhaustive test set in order to adequately test a program.

3 . Monotonicity Property

"If a test set *T* is adequate for program *P*, and if *T* is equal to, or a subset of *T'*, then *T'* is adequate for program *P*."

4 . Inadequate Empty Set

"An empty test set is not an adequate test for any program". If a program is not tested at all, a tester cannot claim it has been adequately tested!

5 . Anti-extensionality Property

"There are programs *P* and *Q* such that *P* is equivalent to *Q*, and *T* is adequate for *P*, but *T* is not adequate for *Q*". Just because two programs are *semantically* equivalent does not mean we should test them the same way. Their implementations (code structure) may be very different.

6 . General Multiple Change Property

"There are programs *P* and *Q* that have the syntactic equivalence, and there is a test set *T* such that *T* is adequate for *P*, but is not adequate for *Q*". Two programs are the same shape if one can be transformed into the other by applying

- **(i)** replace relational operator *r*1 in a predicate with relational operator *r*2;
- **(ii)** replace constant *c*1 in a predicate of an assignment statement with constant *c*2;
- **(iii)** Replace arithmetic operator *a*1 in an assignment statement with arithmetic operator *a*2.

7 . Anti-decomposition Property

"There is a program *P* and a component *Q* such that *T* is adequate for *P*, *T'* is the set of vectors of values that variables can assume on entrance to *Q* for some *t* in *T*, and *T'* is not adequate for *Q*". Although an encompassing program has been adequately tested, it does not follow that each of its components parts has been properly tested.

Eg. a routine that has been adequately tested in one environment may not have been adequately tested to work in another environment, the environment being the enclosing program.

8 . Anti-composition Property

"There are programs *P* and *Q*, and test set *T*, such that *T* is adequate for *P*, and the set of vectors of values that variables can assume on entrance to *Q* for inputs in *T* is adequate for *Q*, but *T* is not adequate for *P; Q*

(the composition of *P* and *Q*)". Adequately testing each individual program component in isolation does not necessarily mean that we have adequately tested the entire program (the program as a whole).

9 . Renaming Property

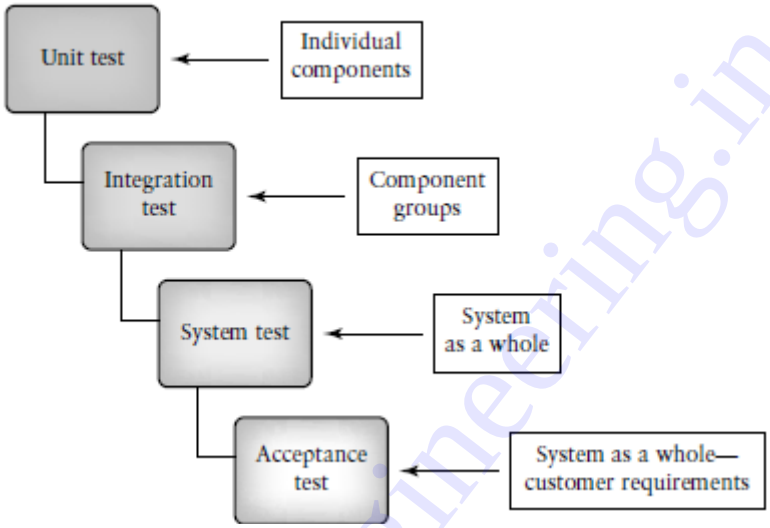
"If *P* is a renaming of *Q*, then *T* is adequate for *P* only if *T* is adequate for *Q*. *A*n inessential change in a program such as changing the names of the variables should not change the nature of the test data that are needed to adequately test the program.

10. Complexity Property

"For every *n*, there is a program *P* such that *P* is adequately tested by a size *n* test set, but not by any size *n*-1 test set." This means that for every program, there are other programs that require more testing.

3.1 THE NEED FOR LEVELS OF TESTING

Execution-based software testing, especially for large systems, is usually carried out at different levels. In most cases there will be 3–4 levels, or major phases of testing: unit test, integration test, system test, and some type of acceptance test as shown in figure. Each of these may consist of one or more sublevels or phases. At each level there are specific testing goals. For example,



Levels of Testing

- At **unit test** a single component is tested. A principal goal is to detect functional and structural defects in the unit.
- At the **integration level** several components are tested as a group, and the tester investigates component interactions.
- At the **system level** the system as a whole is tested and a principle goal is to evaluate attributes such as usability, reliability, and performance.

- The testing process begins with the smallest units or components to identify functional and structural defects. Both white and black box test strategies can be used for test case design at this level.
- After the individual components have been tested, and any necessary repairs made, they are integrated to build subsystems and clusters. Testers check for defects and adherence to specifications.
- System test begins when all of the components have been integrated successfully. It usually requires the bulk of testing resources. At the system level the tester looks for defects, but the focus is on evaluating performance, usability, reliability, and other quality-related requirements.
- During acceptance test the development organization must show that the software meets all of the client's requirements. Very often final payments for system development depend on the quality of the software as observed during the acceptance test.

3.2 UNIT TEST

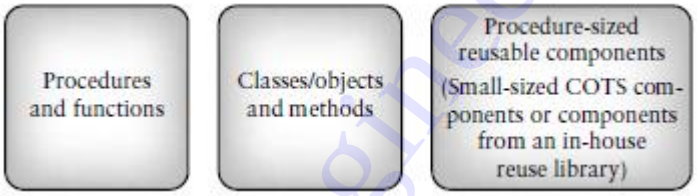
A unit is the smallest possible testable software component.

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.

Examples

- ✓ a function or procedure implemented in a procedural (imperative) programming language.
- ✓ method and the class/object
- ✓ a simple module retrieved from an in-house reuse library



Some components suitable for Unit testing

Unit Testing Advantages

- Easier to design, execute, record, and analyze test results.
- Easier to locate and repair since only the one unit is under consideration.

Unit Test: The Need for Preparation

The ***principal goal for unit testing*** is insure that each individual software unit is functioning according to its specification.

The unit should be tested by an independent tester (someone other than the developer) and the test results and defects found should be recorded as a part of the unit history (made public). Each unit should also be reviewed by a team of reviewers, preferably before the unit test. Some developers also perform an informal review of the unit.

To implement best practices it is important to ***plan for, and allocate resources*** to test each unit.

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

- ➢ Plan the general approach to unit testing;
- ➢ Design the test cases, and test procedures (these will be attached to the test plan);
- ➢ Define relationships between the tests;
- ➢ Prepare the auxiliary code necessary for unit test.

3.2.1 UNIT TEST PLANNING

A general unit test plan should be prepared. It should be developed in conjunction with the master test plan and the project plan for each project. ***Documents that provide inputs for the unit test plan are the project plan, as well the requirements, specification, and design documents that describe the target units***.

Components of a unit test plan are described in detail the *IEEE Standard for Software Unit Testing*.

Phase 1: Describe Unit Test Approach and Risks

In this phase of unit testing planning the test planner:

- ✓ identifies test risks;
- ✓ describes techniques to be used for designing the test cases for the units;
- ✓ describes techniques to be used for data validation and recording of test results;
- ✓ describes the requirements for test harnesses and other software that interfaces with the units to be tested,

During this phase the planner also identifies completeness requirements—what will be covered by the unit test and to what degree (states, functionality, control, and data flow patterns).

The planner also identifies termination conditions for the unit tests. This includes coverage requirements, and special cases. Special cases may result in abnormal termination of unit test (e.g., a major design flaw).

The planner estimates resources needed for unit test, such as hardware, software, and staff, and develops a tentative schedule under the constraints identified at that time.

Phase 2: Identify Unit Features to be tested

This phase requires information from the unit specification and detailed design description. The planner determines which features of each unit will be tested.
Example: functions, performance requirements, states and state transitions, control structures, messages, and data flow patterns.
If some features will not be covered by the tests, they should be mentioned and the risks of not testing them be assessed.

Phase 3: Add Levels of Detail to the Plan

In this phase the planner refines the plan as produced in the previous two phases. The planner adds new details to the approach, resource, and scheduling portions of the unit test plan.

Example, existing test cases that can be reused for this project can be identified in this phase.

The planner must be sure to include a description of how test results will be recorded. Test-related documents that will be required for this task, for example, test logs, and test incident reports, should be described, and references to standards for these documents provided.

3.2.2 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).

Test case data should be tabularized for ease of use, and reuse.
The ***components of a test case*** can be arranged into a semantic network with parts, Object_ID, Test_Case_ID, Purpose, and List_of_Test_Case_Steps. Each of these items has component parts.

As part of the unit test design process, developers/testers should also describe the relationships between the tests. Test suites can be defined that bind related tests together as a group.

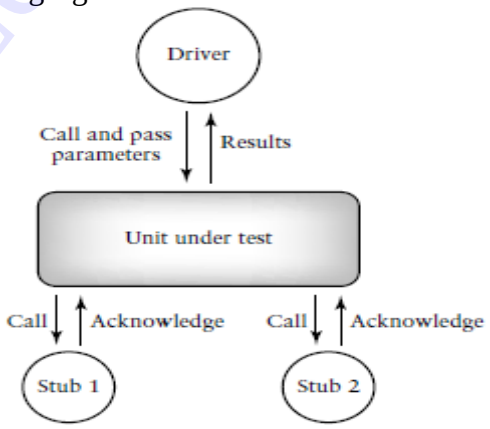
Test case design at the unit level can be based on use of the black and white box test design strategies.
Both of these approaches are useful for

- designing test cases for functions and procedures.
- designing tests for the individual methods (member functions) contained in a class.

Considering the relatively small size of a unit, it makes sense to focus on white box test design for procedures/functions and the methods in a class.

3.2.3 THE TEST HARNESS

In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world. Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit. The role is of the test harness is shown in the following figure.



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.

Drivers and ***stubs*** can be developed at several levels of functionality. For example,

If modules D,E & F,G are the lowest module which is unit tested. Module B & C are not yet developed. The functionality of these modules is that, it calls the modules D, E & F, G. Since B and C are not yet developed, we would need some program or a "stimulator" which will call the D,E& F,G modules. These stimulator programs are called **Drivers:** *dummy programs which are used to call the functions of the lowest module in case when the calling function does not exists.*

In this context, if testing starts from Module A and lower modules B and C are integrated one by one. Now here the lower modules B and C are not actually available for integration. So in order to test the top most modules A, we develop "**Stubs**": *a snippet which accepts the inputs / requests from the top module and returns the results/ response*. This way, in spite of the lower modules do not exist, we are able to test the top module.

3.2.4 RUNNING THE UNIT TESTS AND RECORDING RESULTS

Unit tests can begin when

- the units becomes available from the developers
- the test cases have been designed and reviewed, and
- the test harness, and any other supplemental supporting tools, are available.

The testers then proceed to run the tests and record results. The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format such as shown in following table. These forms can be included in the ***test summary report***.

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 results

It is very important for the tester at any level of testing to carefully record, review, and check test results. The tester must determine from the results whether the unit has passed or failed the test. If the test is failed, the ***nature of the problem*** should be recorded in what is sometimes called a ***test incident report***.

When a unit fails a test there may be several reasons for the failure. The most likely reason for the failure is a fault in the unit implementation (the code).

- ✓ a fault in the test case specification (the input or the output was not specified correctly);
- ✓ a fault in test procedure execution (the test should be rerun);
- ✓ a fault in the test environment (perhaps a database was not set up properly);
- ✓ a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

The ***causes of the failure*** should be recorded in a ***test summary report***, which is a summary of testing activities for all the units covered by the unit test plan.

Ideally, when a unit has been completely tested and finally passes all of the required tests it is ready for integration. Under some circumstances a unit may be given a conditional acceptance for integration test. This may occur when the unit fails some tests, but the impact of the failure is not significant with respect to its ability to function in a subsystem. Units with a conditional pass must eventually be repaired.

Finally, the tester should insure that the test cases, test procedures, and test harnesses are preserved for future reuse.

3.3 INTEGRATION TESTS (Type of testing and Phase of testing)

- ✓ A ***system*** is made up of multiple components or modules that can comprise hardware and software.
- ✓ ***Integration*** is defined as the set of interactions among components.
- ✓ ***Integration testing*** is testing the interaction between the modules and interaction with other systems externally.

The ***need for an Integration Testing*** is to make sure that your components satisfy the following requirements: Functional; Performance; Reliability.

Integration testing as a type of testing : Integration testing means testing of interfaces.

- ➢ ***Internal interfaces*** are those that provide communication across two modules within a project or product, internal to the product, and not exposed to the customer or external developers.
- ➢ ***External interfaces*** are those that are visible outside the product to third party developers and solution providers. A method of achieving interfaces is by providing Application Programming Interfaces (API)

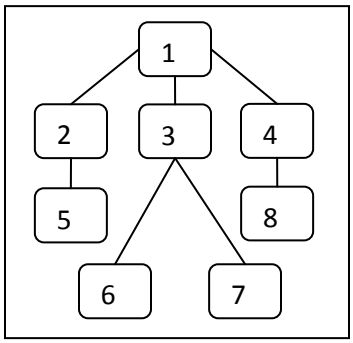
Not all interactions between the modules are known and explained through interfaces. ***Explicit interfaces*** are documented interfaces. ***Implicit interfaces*** are those which are known internally to the software engineers but are not documents
In situations where architecture or design documents do not clear by explain all interfaces among components, some additional test cases are generated and included with others and this approach is termed as ***gray box testing***.

The order in which the interfaces are tested are categorized as follows,

- **Bottom-up** integration: The piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system.
- **Top-down** integration: The breaking down of a system to gain insight into its compositional sub-systems.
- Bi-directional / **Sandwich** integration

Top-Down Integration

Assume a new product where components become available one after another in the order of component numbers. The integration starts with testing the interface between C1 and C2. All interfaces shown in figure covering all the arrows have to be tested together. The tested order is given in table.



Step	Interfaces tested (BFS)
1	1-2
2	1-3
3	1-4
4	1-2-5
5	1-3-6
6	1-3-6-(3-7)
7	(1-2-5)-(1-3-6-(3-7)
8	1-4-8
9	(1-2-5)-(1-3-6-(3-7)-(1-4-8)

When one or two components are added to the product in each increment, the integration testing method pertains to only to those new interfaces that are added only. If an addition has an impact on the functionality of component 5, then integration testing for the new release needs to include only the steps, 4, 7 and 9. To optimize the number of steps, steps 6 and 7, steps 8 and 9 can be combined and executed in single step.

If a component at a higher level requires a modification every time a module gets added to the bottom, then for each component addition integration testing ***needs to be repeated*** starting from step 1. The orders in which the interfaces are tested may follow a ***depth first approach*** or ***breadth first approach.***

Advantages of Top-down Testing

- Drivers do not have to be written when top down testing is used.
- It provides early working module of the program and so design defects can be found and corrected early.

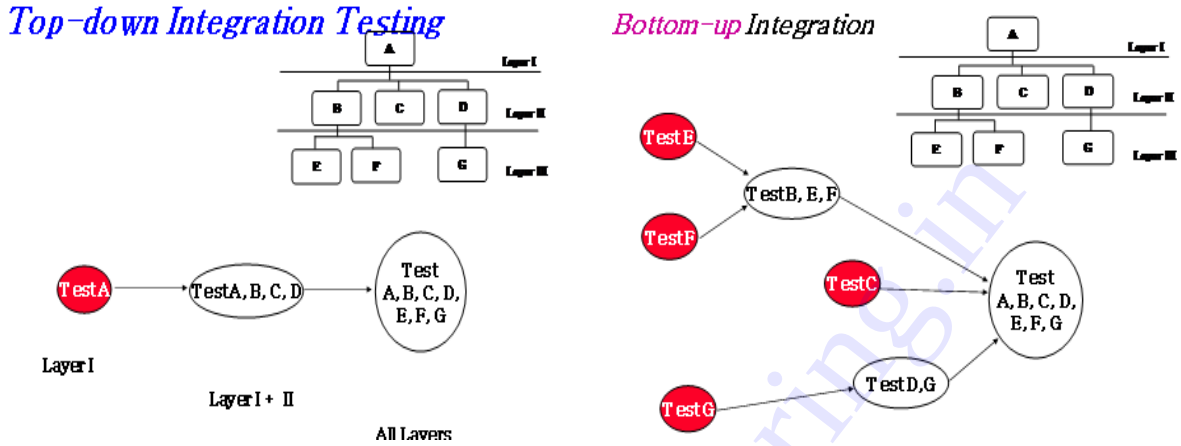
Disadvantages of Top-down Testing

- Stubs have to be written with utmost care as they will simulate setting of output parameters.
- It is difficult to have other people or third parties to perform this testing, mostly developers will have to spend time on this.

Bottom-up Integration

In this approach, the components for a new product development become available in reverse order, start from the bottom.

- Double arrow denotes both the logical flow of components (top to bottom) and integration approach (bottom to up). That means the logical flow of the product can be different from the integration path. The tested order is given in table.



Step	Interfaces tested
1	5, 6, 7, 8
2	2-5, 3-6, (3-6)-3-7, 4-8
8	1-(2-5, 3-6, (3-6)-3-7, 4-8)

Advantages of Bottom-up Testing

- Behavior of the interaction points is crystal clear, as components are added in the controlled manner and tested repetitively.
- Appropriate for applications where bottom up design methodology is used.

Disadvantages of Bottom-up Testing

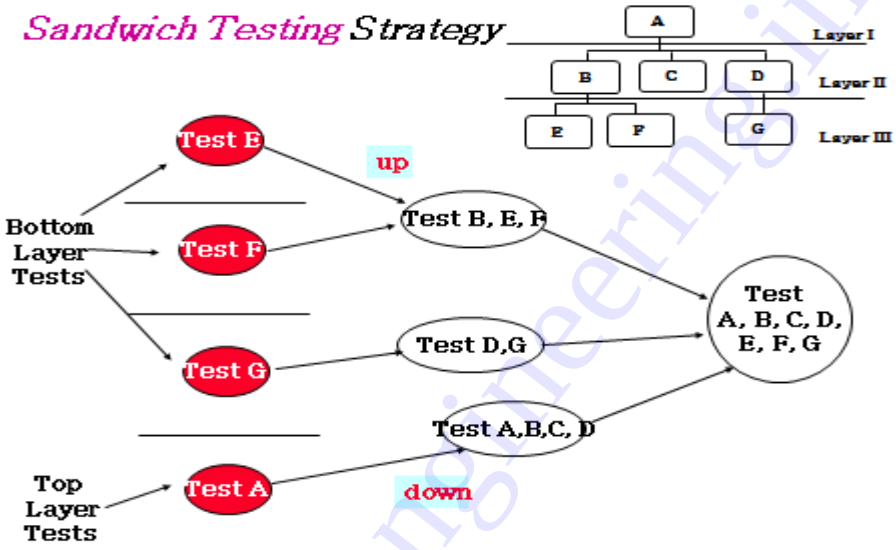
- Writing and maintaining test drivers is more difficult than writing stubs.
- This approach is not suitable for the software development using top-down approach.

Bi-directional Integration

It is a combination of the above two approaches. The individual components (1,2,3,4,5) are tested separately. Then the bi-directional integration is performed with the use of stubs and drivers. After the functionality of these integrated components is tested, the stubs and drivers are discarded. Once components 6,7,8 are available, this method then focuses only on those components, as these are the components that are new and need focus. This approach is also called ***sandwich integration***. The tested order is given in table

Step	Interfaces tested
1	6-2
2	7-3-4
3	8-5
4	(1-6-2)-(1-7-3-4)-(1-8-5)

- Combines top-down strategy with bottom-up strategy
- *The system is view as having three layers*
  - **A target layer in the middle**
  - **A layer above the target**
  - **A layer below the target**
  - **Testing converges at the target layer**
- How do you select the target layer if there are more than 3 layers?
  - **Heuristic: Try to minimize the number of stubs and drivers**



Steps in Integration Testing (Summary)

1. Based on the integration strategy, *select a component* to be tested. Unit test all the ***classes*** in the component.
2. Put selected ***component*** together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Do *functional testing:* Define test cases that exercise all ***uses cases*** with the selected component
4. Do *structural testing:* Define test cases that exercise the selected ***component***
5. Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

3.3.1 DESIGNING INTEGRATION TESTS

Integration tests for procedural software can be designed using a black or white box approach. The tester needs to insure the parameters are of the correct type and in the correct order. The tester must also insure that once the parameters are passed to a routine they are used correctly.

Testers must insure that test cases are designed so that all modules in the structure chart are called at least once, and all called modules are called by every caller.

Coverage requirements for the internal logic of each of the integrated units should be achieved during unit tests. When units are integrated and subsystems are to be tested as a whole, new tests will have to be designed to cover their functionality and adherence to performance and other requirements.
Sources for development of black box or functional tests at the integration level are,

- The requirements documents and
- The user manual.

Testers need to work with requirements analysts to insure that the requirements are,

- testable,
- accurate, and
- complete.

Black box tests should be developed to insure proper functionally and ability to handle subsystem stress. For example, in a transaction-based subsystem the testers want to determine the limits in number of transactions that can be handled.

Integration testing of clusters of classes also involves building test harnesses which in this case are special classes of objects built especially for testing. Whereas,

- in class testing we evaluated intra-class method interactions,
- at the cluster level we test interclass method interaction as well.

Unlike procedural-oriented systems, integration for object-oriented systems usually does not occur one unit at a time. A group of cooperating classes is selected for test as a cluster.

If developers have used the Coad and Yourdon's approach, then a subject layer could be used to represent a cluster.

Jorgenson et al. have reported on a notation for a cluster that helps to formalize object-oriented integration.

A method-message path is described as a sequence of method executions linked by messages. An atomic system function is an input port event (start event) followed by a set of method messages paths and terminated by an output port event (system response). Murphy et al. define clusters as classes that are closely coupled and work together to provide a unified behavior. Some examples of clusters are

- Groups of classes that produce a report, or monitor and control a device.

Integration testing as a phase of testing

Integration testing as a phase involves different activities and different types of testing have to be done in that phase. This is a testing phase that should ensure completeness and coverage of testing for functionality. To achieve this, the focus should not only be on planned test case execution but also on unplanned testing, which is termed as "ad hoc testing". This approach helps in locating some problems which are difficult to find by test teams but also difficult to imagine in the first place.

3.3.2 INTEGRATION TEST PLANNING

Integration test must be planned. Planning can begin when high-level design is complete so that the system architecture is defined. Other documents relevant to integration test planning are,

- the requirements document,
- the user manual, and
- usage scenarios.

These documents contain,

- structure charts,
- state charts,
- data dictionaries,
- cross-reference tables,
- module interface descriptions,
- data flow descriptions,
- messages and event descriptions

The strategy for integration should be defined. For procedural-oriented system the order of integration of the units should be defined. Consider the fact that the testing objectives are to assemble components into subsystems and to demonstrate that the subsystem functions properly with the integration test cases. For object-oriented systems a working definition of a cluster or similar construct must be described, and relevant test cases must be specified.

A detailed description of a Cluster Test Plan includes the following items:

- clusters this cluster is dependent on;
- a natural language description of the functionality of the cluster to be tested;
- list of classes in the cluster;
- a set of cluster test cases.

One of the goals of integration test is to build working subsystems, and then combine these into the system as a whole. When planning for integration test the planner selects subsystems to build based upon the requirements and user needs. Developers may want to show clients that certain key subsystems have been assembled and are minimally functional.

3.3.3 SCENARIO TESTING

When the functionality of different components are combined and tested together for a sequence of related operations, they are called scenarios. Scenario testing is a planned activity to explore different usage patterns and combine them into test case called scenario test cases.

A set of realistic user activities that are used for evaluation the product. Methods to evolve scenarios are,

- System scenarios
- Role based scenarios

System scenarios

The set of activities used for scenario testing covers several components in the system. The various approaches are,

1. **Story line** : Develop a story line that combines various activities of the product that may be executed by an end user. Eg. User enters his office, logs into system, checks mail, responds to mail, compiles programs, performs unit testing etc.
2. **Life cycle / State transition** : Consider an object, derive the different transitions / modifications that happen to the object, and derive scenarios to cover them. Eg. Account open, deposit money, perform withdrawal, calculate interest etc. Different transformations applied to the object "money" becomes different scenarios.
3. **Deployment / Implementation stories from customer** : Develop a scenario from a known customer deployment / implementation details and create a set of activities by various users in that implementation.
4. **Battle ground** : Create some scenarios to justify that "the product works" and some scenarios to "try and break the system" to justify "the product does not work"

Any activity in the scenario is always a continuation of the previous activity, and depends on or is impacted by the results of previous activities. Considering only one aspect would make scenarios ineffective. A right mix of scenarios using the various approaches explained is very critical for the effectiveness of scenario testing.
Coverage is always a big question with respect to functionality in scenario testing. By using a simple technique, some comfort feeling can be generated on the coverage of activities by scenario testing.

End-user activity	Frequency	Priority	Applicable environment	No. of items covered
Login to application	High	High	W2000, 2003, XP	10
Create an object	High	Medium	W2000, XP	7
Modify parameters	Medium	Medium	W2000, XP	5
List object parameter	Low	Medium	W2000, 2003, XP	3
Compose email	Medium	Medium	W2000, XP	6
Attach files	Low	Low	W2000, XP	2
Send composed mail	High	High	W2000, XP	10

It is clear that important activities have been very well covered by set of scenarios in the system scenario test.

Role based / Use case scenarios

Use case scenario is a stepwise procedure on how a user intends to use a system, with different user roles and associated parameters. It can include stories, pictures and deployment details.

A use case can involve several roles or class of users who typically perform different activities based on the role. Use can scenarios term the users with different roles as **actors**. What the product should do for a particular activity is termed as **system behavior**. Users with a specific role to interact between the actors and the system are called **agents**.

Eg. Cash withdrawal from a bank

- Customer fill up a cheque
- Gives it to an official
- Official verifies the balance
- Gives required cash

Customer – actor, clerk – agent, system response – computer gives balance in account.

This way of describing different roles in test cases helps in testing the product without getting into the details of the product.

- **Actor** : need concerned only about getting cash. Not concerned about what official is doing and what command he uses to interact with computer
- **Agent** : not concerned about the logic of how computer works. Concerned about whether he will get money or not.

Testers using the use case model, with one person testing the actions and other person testing the system response, complement each other's testing as well as testing the business and the implementation aspect of the product at the same time.

In a completely automated system involving the customer and the system, use cases can be written without considering the agent portion.

Actor	System response
Users likes to withdraw cash and inserts card in ATM	Request for password or PIN
User fills password or PIN	Validate the password or PIN Give menu for process
User selects account type	Ask user for amount to withdraw
User fills the amount of case required	Check availability of funds Update account balance Prepare receipt Dispense cash
Retrieve cash from ATM	Print receipt

This way of documenting a scenario and testing makes it simple and also makes it realistic for customer usage.

3.3.4 DEFECT BASH ELIMINATION

Defect bash is an ad hoc testing where people performing different roles in an organization test the product together at the same time. What is to be tested is left to an individual's decision and creativity. They can try some operations which are beyond the product specifications.

Advantages

- Enabling people "*cross boundaries and test beyond assigned areas*".
- Different people performing different roles together in the organization – "*Testing is not for tester's alone*".
- Letting everyone to use the product before delivery.

- Bringing fresh pairs of eyes to uncover new defects – "*Fresh eyes have less bias*".
- Brining people of different levels understanding to test the product together randomly – "*users of software are not same*".
- Let testing don't wait for lack of / time taken for documentation – "*Does testing wait till all documentation is done*".

Even though it is said that defect bash is an ad hoc testing, not all activities of defect bash are unplanned. ***All the activities are planned activities***, except for what to be tested. The involved steps are,

- **Choosing the frequency and duration of defect bash**
  - ✓ Frequent defect bash → incur low return on investment
  - ✓ Too few defect bash → may not meet the objective of finding all defects.
  - ✓ Duration optimization → big saving
  - ✓ Small duration → amount of testing that is done may not meet the objective.
- **Selecting the right product build**
  - ✓ Regression tested build → ideal as all new features and defect fixes would have been already tested
  - ✓ Intermediate build (code functionality is evolving) / Untested build → make the purpose and outcome of a defect bash ineffective
- **Communicating the objective of defect bash**

  The objective should be,
  - To find a large number of uncovered defects
  - To find out system requirements (cpu, memory, disk etc)
  - To find the non-reproducible or random defects

  Defects that a test engineer would find easily should not be the objective.
- **Setting up and monitoring the lab**

  Finding the right configuration, resources (hardware, software, set of people) should be the plan before starting defect bash.

  The majority of defect bash fail due to inadequate hardware, wrong software configurations and perceptions related to performance and scalability of the software.
  - ➢ The defects that are in the product, as reported by the users → ***functional defects***
  - ➢ The defects that are revealed while monitoring the resources (memory leak, long turnaround time, missed requests, high impact etc.) → ***non-functional defects***.
  - ➢ Defect bash test is a unique testing method which can bring out both these defects.
- **Taking actions and fixing issues**

  Many defects could be duplicate defects. It is difficult to solve all the problems if they are taken one by one and fixed in code. The defects need to be classified into issues at a higher level, so that a similar outcome can be avoided in future defect bashed.

  There could be one defect associated with an issue and there could be several defects that can be called as an issue.
- **Optimizing the effort involved in defect bash**

  An approach to reduce the defect bash effort is to conduct "**micro level**" defect bashes before conducting one on a large scale.

To prevent component level defect emerging during integration testing, a micro level defect bash can also be done to unearth feature level defects before integration testing.
Here, the defect bash can be classified into,

- Feature / component defect bash
- Integration defect bash
- Product defect bash

Example.

3 product defect bashed conducted in 2 hours with 100 people.
Total effort involved is 3 * 2 * 100 = 600 person hours.
If feature / component test team and integration test team, that has 100 people each, conduct 2 rounds of micro level bashes, which can find out 1/3 of defects, then effort saving is 20%.

Total effort involved in 2 rounds of product bashes = 400 man hours
Effort involved in 2 rounds of feature bash = 2 * 2 * 10 – 40
Effort involved in 2 rounds of integration bash = 2 * 2 * 10 – 40

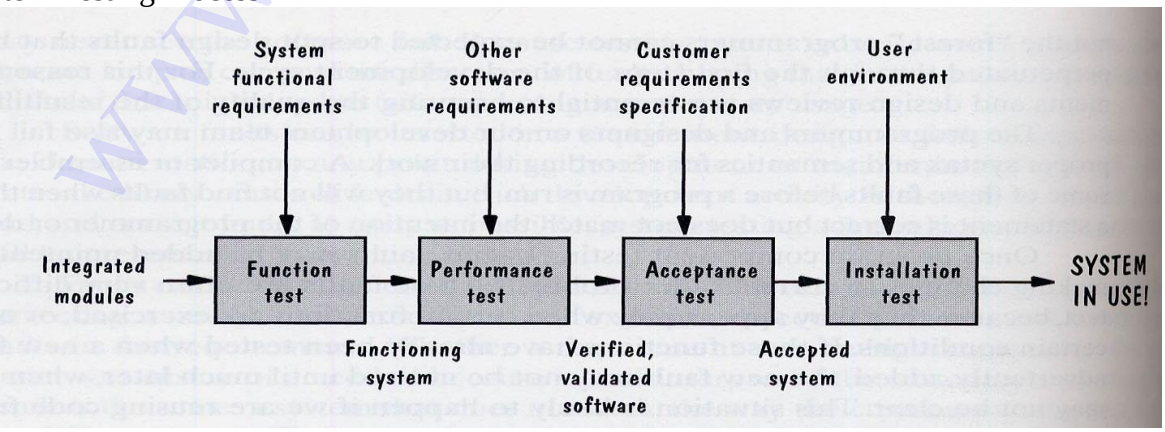
Effort saved = 600 – (A+B+C) = 600 – 480 = 120 person hours, or 20%.

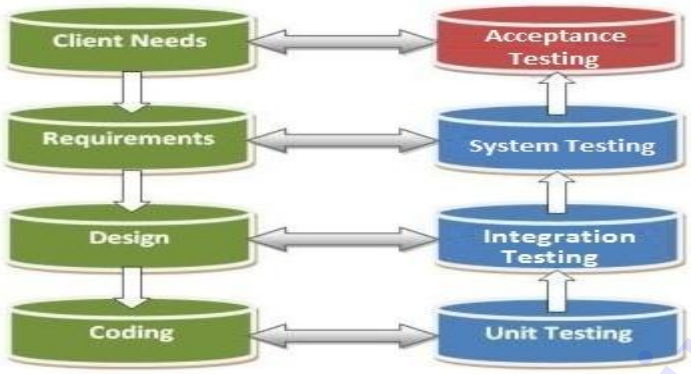
3.4 SYSTEM TESTING

The testing conducted on the complete integrated products and solutions to evaluate system compliance with specified requirements on functional and non-functional aspects is called system testing.

A system is complete set of integrated components that together deliver product functionality and features. System testing helps in uncovering the defects that may not be directly attributable to a module or an interface. System testing brings out issues that are fundamental to design, architecture and code of whole product.

System Testing Process





Functional testing Vs non-functional testing

Testing aspects	Functional testing	Non-functional testing
Involves	Product features and functionality	Quality factors
Tests	Product behavior	Behavior and experience
Result conclusion	Simple steps written to check expected results	Huge data collected and analyzed
Results varies due to	Product implementation	Product implementation, resources and configurations
Testing focus	Defect detection	Qualification of product
Knowledge required	Product and domain	Product, domain, design, architecture, statistical skills
Failures normally due to	Code	Architecture, design and code
Testing phase	Unit, component, integration, system	system

3.5 ACCEPTANCE TESTING

Acceptance testing is a phase after system testing that is normally done by the customers or representative of the customer. Acceptance test is performed by the client, not by the developer. Acceptance test cases are normally small in number and are not written with the intention of finding defects. However, the purpose of this test is to enable customers and users to determine if the system built really meets their needs and expectations. Acceptance testing is done by the customer or by the representative of the customer to check whether the product is ready for use in the real-life environment.

Acceptance tests are written to execute near real-life scenarios.

Acceptance Criteria.

1. Product acceptance
2. Procedure acceptance
3. Service level agreements

Selection test cases for Acceptance Testing

- ❖ End-to-end functionality verification
- ❖ Domain tests
- ❖ User scenario tests
- ❖ Basic sanity tests
- ❖ New functionality
- ❖ A few non-functionality
- ❖ Tests pertaining to legal obligations and service level agreements
- ❖ Acceptance test data

METHOD

Usually, Black Box Testing method is used in Acceptance Testing. Testing does not normally follow a strict procedure and is not scripted but is rather ad-hoc.

TASKS

- Acceptance Test Plan
  - ➔ Prepare → Review → Rework → Baseline
- Acceptance Test Cases/Checklist
  - ➔ Prepare → Review → Rework → Baseline
- Acceptance Test
  - Perform

When is it performed?

Acceptance Testing is performed after System Testing and before making the system available for actual use.

Who performs it?

- ***Internal Acceptance Testing*** (Also known as **Alpha Testing**) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support.
- ***External Acceptance Testing*** is performed by people who are not employees of the organization that developed the software.
  - *Customer Acceptance Testing* is performed by the customers of the organization that developed the software. They are the ones who asked the organization to develop the software. [This is in the case of the software not being owned by the organization that developed it.]
  - *User Acceptance Testing* (Also known as Beta Testing) is performed by the end users of the software. They can be the customers themselves or the customers' customers.

Types of Acceptance testing

- **Benchmarking:** a predetermined set of test cases corresponding to typical usage conditions is executed against the system
- **Pilot Testing:** users employ the software as a small-scale experiment or in a controlled environment

- **Alpha-Testing:** pre-release closed / in-house user testing
- **Beta-Testing:** pre-release public user testing
- **Parallel Testing:** old and new software are used together and the old software is gradually phased out.

3.6 PERFORMANCE TESTING.

The testing performed to evaluate the response time, throughput, and utilization of the system, to execute its required functions in comparison with different versions of the same products or a different competitive product is called ***performance testing***.

It is done ***to ensure*** that a product,

- Processes the required number of transactions in any given interval (throughput)
- Is available and running under different load conditions (availability)
- Delivers worthwhile return on investment for the resources and deciding what kind of resources are needed for the product for different load conditions (load capacity)
- Is comparable to and better than that of the competitors for different parameters.

Methodology for performance testing

Collecting requirements

- Performance compared to the previous release of the same product
- Performance compared to the competitive products
- Performance compared to absolute numbers derived from actual need
- Performance numbers derived from architecture and design
- Example

Transaction	Expected response time	Loading pattern / throughput	Machine configuration
ATM cash withdrawal	2 sec	Up to 10,000 simultaneous access by users	Pentium IV / 512 MB RAM / broadband network
ATM cash withdrawal	40 sec	Up to 10,000 simultaneous access by users	Pentium IV / 512 MB RAM / dialup network
ATM cash withdrawal	4 sec	More than 10,000 but below 20,000 simultaneous access by users	Pentium IV / 512 MB RAM / broadband network

Writing test cases

- List of operations or business transactions to be tested
- Steps for executing those operations or transactions
- List of product, OS parameters that impact the performance testing
- Resource and their configuration (network, hardware)
- The expected results (response time, throughput, latency)

Automating performance test cases

- Performance testing is repetitive
- Performance test cases cannot be effective without automation
- The results need to be accurate

Sample configuration performance test

Transaction	Number of users	Test environment
Querying ATM account balance	20	RAM 512 MB, P4 Dual processor; OS – Windows NT server
ATM cash withdrawal	20	RAM 128 MB, P4 Single processor; OS – Windows 98
ATM user profile query	40	RAM 256 MB, P3 Quad processor; OS – Windows 2000

Different levels of Performance Testing

- **Stress Testing** : Stress limits of system
- **Volume testing** : Test what happens if large amounts of data are handled
- **Configuration testing** : Test the various software and hardware configurations
- **Compatibility test** : Test backward compatibility with existing systems
- **Timing testing** : Evaluate response times and time to perform a function
- **Security testing** : Try to violate security requirements
- **Environmental test** : Test tolerances for heat, humidity, motion
- **Quality testing** : Test reliability, maintain- ability & availability
- **Recovery testing** : Test system's response to presence of errors or loss of data
- **Human factors testing** : Test with end users.

Analyzing performance test results

Performance tuning

- Tuning the product parameters
- Tuning the operating system and parameters

Performance benchmarking

- Identifying the transactions / scenarios and the test configuration
- Comparing the performance of different products
- Tuning the parameters of the products being compared fairly to deliver the best performance

Tools for performance testing

➔ Functional performance tool
- ✓ WinRunner from Mercury
- ✓ QA Partner from Compuware
- ✓ Silktest from Segue

→ Load testing tools
- * Load Runner from Mercury
- * QA Load from Compuware
- * Silk Performer from Segue

3.7 REGRESSION TESTING

When a bug is fixed by the development team than testing the other features of the applications which might be affected due to the bug fix is known as **regression testing**. **Regression testing** is always done to verify that modified code does not break the existing functionality of the application and works within the requirements of the system.

There are mostly two strategies to **regression testing**,

1) to run all tests and
2) always run a subset of tests based on a test case prioritization technique.

Regression testing is the re-testing of features to make safe that features working earlier are still working fine as desired. It is executed when any new build comes to QA, which has bug fixes in it or during releasing cycles (Alpha, Beta or GA) to originate always the endurance of product.

There are two types of regression testing → Regular regression testing, Final regression testing

Regular regression testing is done between test cycles to ensure that the defect fixes that are done and the functionality that were working with the earliest test cycles continue to work.
It is necessary to perform regression testing when,

1. A reasonable amount of initial testing is already carried out
2. A good number of defect have been fixed
3. Defect fixes that can produce side-effects are taken care of
   Final regression testing is done to validate the final build before release.

Methodologies used for regression testing.

1. Performing an initial "Smoke" or "Sanity" test
   - ❖ Identifying the basic functionality that a product must satisfy
   - ❖ Designing test cases to ensure that these basic functionality work and packaging them into a smoke test suite
   - ❖ Ensuring that every time a product is built, this suite is run successfully before anything else is run.
2. Understanding the criteria for selecting the test cases
   - ➢ Include test cases that have produced the maximum defect in the past
   - ➢ Include test cases in which problems are reported
   - ➢ Include test cases that test the end-to-end behavior of the application or the product
   - ➢ Include test cases to test the positive test condition
   - ➢ Includes the area which is highly visible to the users.
3. Classifying the test cases into different priorities
   - ✓ Priority 0: these test cases can be called sanity test cases which check basic functionality and are run for accepting the build of further testing.
   - ✓ Priority 1: Uses the basic and normal setup and these test cases deliver high project value to both development team and to customers.

4. A methodology for selecting test cases
    - Case 1: If the criticality and impact of the defect fixes are low, then it is enough that a test engineer selects a few test cases from test case database
    - Case 2: If the criticality and impact of the defect fixes are medium, then it is needed to execute all priority-0 and priority-1 test cases.
    - Case 3: If the criticality and impact of the defect fixes are high, then it is needed to execute all priority-0 and priority-1 and a carefully selected subset of priority-2.
    - Alternative methodologies
        - Regress all
        - Priority based regression
        - Regress changes
        - Random regression
        - Context based dynamic regression
5. Resetting the test cases for test execution
6. Concluding the results of a regression cycle

Best Practices in Regression Testing

Practice 1 → Regression can be used for all types of releases
Practice 2 → Mapping defect identifiers with test cases improves regression quality
Practice 3 → Create and execute regression test bed daily
Practice 4 → Ask your best test engineer to select the test cases
Practice 5 → Detect defects, and protect your product from defects and defect fixes

Current result from regression	Previous result	Conclusion	Remarks
FAIL	PASS	FAIL	Need to improve the regression process an code reviews
PASS	FAIL	PASS	This is the expected result of a good regression
FAIL	FAIL	FAIL	Need to analyze why defect fixes are not working
PASS (with a work around)	FAIL	Analyze the workaround and if satisfied mark result as PASS	Workaround also need a good review
PASS	PASS	PASS	This pattern of results give a comfort feeling

Regression Testing Tools

Automated Regression Testing is the testing area where we can automate most of the testing efforts. We run all the previously executed test cases on new build. This means we have test case set available and running these test cases manually is time consuming. We know the expected results so automating these test cases is time saving and efficient regression test method. Extent of automation depends on the number of test cases that are going to remain applicable over the time.

3.8 ADHOC TESTING

There are different variants and types of testing under ad hoc testing.

1. Buddy testing
   - ✓ A developer and tester working as buddies to help each other on testing and in understanding the specifications is called buddy testing.
2. Pair testing
   - ✓ Pair testing is testing done by two testers working simultaneously on the same machine to find defects in the product.
   - ✓ Example : Two people traveling in a car in a new area to find a place, with one driving and other navigating with a map.
   - ✓ Finding new place (like finding defects in the unexplored area of the product) becomes easier as there are two people putting their heads together with specific roles such as navigation and driving assigned between them.
3. Exploratory testing
   - ✓ Example: Driving a car to a place in a new area without a map. Common techniques used are,
   - ✓ Getting a map of the area
   - ✓ Traveling in some random direction to figure out the place
   - ✓ Calling up and asking a friend for the route
   - ✓ Asking for directions by going to a near-by fuel station
   - ✓ Technical equivalences for the above example
4. Iterative testing
   - ✓ Example: Driving a car without a map, trying to count number of hotels in an area. When he reaches a multi-way junction, he may take one road at a time and search for hotels on that road. Then he can go back to the multi-way junction and try a new road. He can continue doing this till all the roads have been explored for counting the hotels.
5. Agile and Extreme testing (XP model)
   - ✓ The different activities involved in XP work flow are as follows,
     Develop user stories; Code; Test; Refactor; Delivery
6. Defect seeding
   - ✓ Defect seeding is a method of intentionally introducing defects into a product to check the rate of its detection and residual defects.
   - ✓ For example, assume that 20 defects that range from critical to cosmetic errors are seeded on a product
   - ✓ Suppose when the test team completers testing it has found 12 seeded defects and 25 original defects.
   - ✓ The total number of defects that may be latent with the product is calculates as follows,
   - ✓ Total latent defects = (Defects seeded / Defects seeded found) * Original defects found.
   - ✓ So the number of estimated defects, based on the above example = (20/12)*25 = 41.67

Characteristics of ad-hoc testing:

- They are always in line with the test objective. However they are certain drastic tests performed with intent to break the system.
- The tester needs to have complete knowledge and awareness about the system being tested. The result of this testing finds bugs that attempts to highlight loopholes of the testing process.
- Also looking at the above two tests, the natural reaction to it would be that – these kind of tests can be performed just once as it's not feasible for a re-test unless there is a defect associated.

Benefits of Ad-hoc testing

- A tester can find more number of defects than in traditional testing because of the various innovative methods they can apply to test the software.
- It's not only restricted to the testing team, but anywhere in SDLC. This would help developers code better and also predict what problems might occur.
- Can be coupled with other testing to get best results which can sometimes cut short the time needed for the regular testing.
- Doesn't mandate any documentation to be done which prevents extra burden on the tester. Tester can concentrate on actually understanding the underlying architecture.
- In cases when there is not much time available to test, this can prove to be very valuable in terms of test coverage and quality

Ad-hoc testing drawbacks:

- Since it's not very organized and there is no documentation mandated, the most evident problem is that the tester has to remember and recollect all the details of the ad-hoc scenarios in memory. This can be even more challenging especially in scenarios where there is a lot of interaction between different components.
- This would also result in not being able recreate defects in the subsequent attempts, if asked for information.
- Since this is not planned/ structured, there is no way to account for the time and effort invested in this kind of testing.
- Ad-hoc testing has to only be performed by a very knowledgeable and skilled tester in the team as it demands being proactive and intuition

Functional System Testing

Functional system testing is performed at different phases and the focus is on product level features. There are two obvious problems. One is ***duplication*** and other one is ***grey area***. ***Duplication*** refers to the same tests being performed multiple times and ***gray area*** refers to certain tests being missed out in all the phases.

1. **Deployment Testing**

The short-term success or failure of a particular product release is mainly assessed on the basis of on how well these customer requirements are met. This type of deployment testing that happens in a product development company to ensure that customer deployment requirements are met is called ***offsite deployment***.

- Deployment testing is also conducted after the release of the product by utilizing the resources and setup available in customer's location. This is a combined effort by the product development organization and the organization trying to use the product. This is called ***onsite deployment***.
  - Online deployment testing is done at two stages.
  - **Stage 1:** Actual data from the live system is taken and similar machines and configurations are mirrored, and the operations from the users are rerun on the mirrored deployment machine.
  - Some use, intelligent recorders to record the transactions that happen on a live system and commit these operations on a mirrored system and then compare the results against the live system.
  - **Stage 2**: the mirrored system is made a live system that runs the new product. Regular backups are taken and alternative methods are used to record the incremental transactions from the time mirrored system became live.

2. **Alpha Testing**

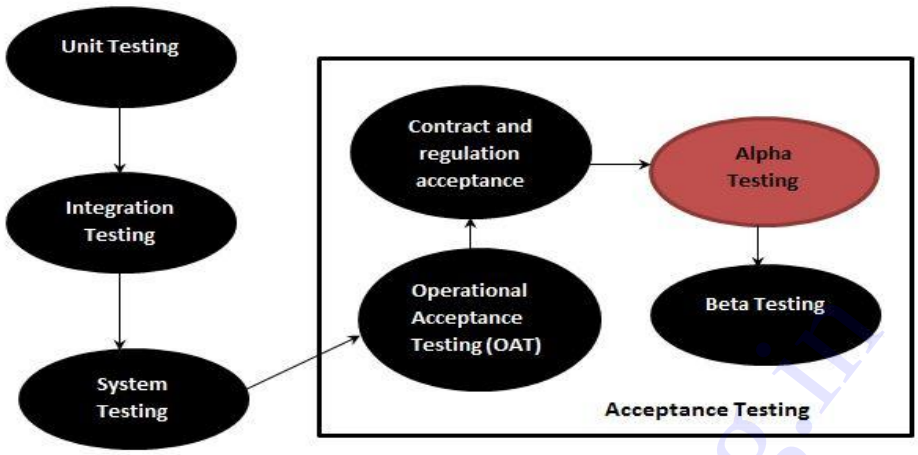
Alpha testing usually comes after system testing and involves both white and black box testing techniques. The company employees test the software for functionality and give the feedback. After this testing phase any functions and features may be added to the software.
Alpha Testing is done to ensure confidence in the product or for internal acceptance testing, alpha testing is done at the developer's site by independent test team, potential end users and stakeholders. Alpha Testing is mostly done for COTS(Commercial Off the Shelf) software to ensure internal acceptance before moving the software for beta testing.

The main features of Alpha testing are:

- outside users are not involved while testing;
- white box and black box practices are used;
- Developers are involved.

How do we run it?

- In the first phase of alpha testing, the software is tested by in-house developers during which the goal is to catch bugs quickly.
- In the second phase of alpha testing, the software is given to the software QA team for additional testing.
- Alpha testing is often performed for Commercial off-the-shelf software (COTS) as a form of internal acceptance testing, before the beta testing is performed.



3. **Beta Testing**

- Delays in product releases and the product not meeting the customer requirements are common. A product rejected by the customer after delivery means a huge loss to the organization.
- There are many **reasons for a product not meeting the customer's requirements** and thus get rejected.
  - ✓ A product not meeting the implicit requirements
  - ✓ Customer's business requirements keep changing constantly and a failure to reflect these changes in the product
  - ✓ Picking up the ambiguous areas and not resolving them.
  - ✓ The understanding of the requirements may be correct but the implementation could be wrong.
  - ✓ Lack of usability and documentation.

The mechanism of sending the product that is under test to the customers and receiving the feedback. This is called ***beta testing***.
Testing done by the potential or existing users, customers and end users at the external site without developers involvement is known as **beta testing**. It is **operation testing** i.e. It tests if the software satisfies the business or operational needs of the customers and end users.

This is a testing stage followed by internal full alpha test cycle. This is the final testing phase where companies release the software for few external user groups outside the company test teams or employees. This initial software version is called as **beta version**.

Beta Software – Preview version of the software released to the public before final release.
Beta Version – Software version releases in public that include almost all of the features but not development complete yet and may still have some errors.
Beta Testers – Testers who work on testing beta version of the software release.

➢ **Activities involved in the beta program** are as follows,

- ✓ Collecting the list of customers and their beta testing requirements
- ✓ Working out a beta program schedule and informing the customers
- ✓ Sending some documents for reading in advance and training customer on product usage
- ✓ Testing the product to ensure it meets "beta testing entry criteria".
- ✓ Sending the beta product to the customer and enable them to carry out their own testing
- ✓ Collecting the feedback periodically from the customers and prioritizing the defects for fixing
- ✓ Responding to customer's feedback with products fixes or documentation changes.
- ✓ Analyzing and concluding whether the beta program met the exit criteria
- ✓ Release the changed version for verification to the same groups
- ✓ Once all tests are complete do not accept any further feature change request for this release
- ✓ Communicate the progress and action items to customers and formally closing the beta program
- ✓ Incorporating the appropriate changes in the product.
- ✓ Remove the beta label and release the final software version

The main features of Beta testing are:

- outside users are involved;
- black box practices are used.

Both alpha and beta testing are very important while checking the software functionality and are necessary to make sure that all users' requirements are met in the most efficient way.

Difference between Alpha and Beta Testing

Alpha Testing	**Beta Testing (Field Testing)**
1. It is always performed by the developers at the software development site.	1. It is always performed by the customers at their own site.
2. Sometimes it is also performed by Independent Testing Team.	2. It is not performed by Independent Testing Team.
3. Alpha Testing is not open to the market and public	3. Beta Testing is always open to the market and public.
4. It is conducted for the software application and project.	4. It is usually conducted for software product.
5. It is always performed in **Virtual Environment**.	5. It is performed in **Real Time Environment**.
6. It is always performed within the organization.	6. It is always performed outside the organization.
7. It is the form of Acceptance Testing.	7. It is also the form of Acceptance Testing.

8. Alpha Testing is definitely performed and carried out at the developing organizations location with the involvement of developers.	8. Beta Testing (field testing) is performed and carried out by users or you can say people at their own locations and site using customer data.
11. It is always performed at the developer's premises in the absence of the users.	11. It is always performed at the user's premises in the absence of the development team.

4. Compliance Testing for Certification, Standards

- A product needs to be certified with the popular hardware, operating system, database, and other infrastructure pieces. This is called ***certification testing***.
- The sale of a product depends on whether it was certified with the popular systems or not.
- This is one type of testing where there is equal interest from the product development organization, the customer, and certification agencies to certify the product.
- The product development organization runs those certification test suites and corrects the problems in the product to ensure that tests are successful.
- Once the tests are successfully run, the results are sent to the certification agencies and they give the certification for the product.
- There are many ***standards*** for each technology area and the product may need to conform to those standards.
- The product development companies select the standards to be implemented at the beginning of the product cycle.
- Some of the standards are evolved by the open community and published as public domain standards.
- Testing the product to ensure that these standards are properly implemented is called ***testing for standards***.

Non–Functional System Testing

1. Scalability Testing

- The objective of scalability testing is to find out the maximum capability of the product parameters.
- The resources that are needed for this kind of testing are normally very high.
- **Example**: finding out how many client machines can simultaneously log in to the server to perform some operations
- Trying to simulate that kind of real-life scalability parameter is very difficult by at the same time very important.
- A high-end configuration is selected and the scalability parameter is increased step by step to reach the maximum capability.
- Testing continues till the maximum capability of a scalable parameter is found out for a particular configuration.
- ***Failures during scalability test include the system not responding, or the system crashing***, etc.
- If ***resources are problem***, they are increased after validating.

- ❖ If ***OS or technology is problem***, the product organization is expected to work with the OS and technology vendors.

- o Scalability testing is performed on different configurations to check the products behavior. For example,
- o If CPU utilization approaches to 100%, then another server is set up to share the load or another CPU is added to the server.
- o If the results are successful, then the tests are repeated for 200 users and more to find the maximum limit for that configuration.
- o If the CPU utilization is 100%, but only for a short time and if for the rest of the testing is remained at say 40%, then there is no point in adding one more CPU. But, the product still has to be analyzed for the sudden spike in the CPU utilization and it has to be fixed.

2. Reliability Testing

Reliability testing is done to evaluate the product's ability to perform its required function under stated conditions for a specified period of time or for a large number of iterations.

Eg: querying a database continuously for 48 hrs, performing login operations 10,000 times.

Reliability of the product testing is entirely different from reliability testing.

Reliability testing refers to testing the product for a continuous period of time. Reliability testing delivers a "reliability tested product" but not a reliable product. The main factor that is taken into account for reliability testing is defects.

Memory leak is a problem that is normally brought out by reliability testing. At the end of repeated operations sometimes the CPU may not get released or disk and network activity may continue. Hence, it is important to collect data regarding various resources used in the system before, during, and after reliability test execution, and analyze those results.

The CPU and memory utilization must be consistent throughout the test execution. If they keep on increasing, other applications on the machine can get affected; the machine may even run out of memory, hand or crash, in which case the machine needs to be restarted.

The following table gives an idea on ***how reliability data can be collected***.
Configuration details: Memory – 1GB; Processors – 2.850 MHz; N/W bandwidth – 100 Mbps.

Test data	25 clients	60 clients	100 clients
Total iterations			
Total fail			
Peak memory utilization (MB)			
Mean memory utilization (MB)			
Peak CPU utilization			
Mean CPU utilization			
Picketer received at server			
Packets sent from server			

Different ways of expressing reliability defects in charts

Mean time between failures → the average time elapsed from between successive product failures. For example, if the product fails, say for every 72 hours.

Failure rate → function that gives the number of failures occurring per unit time

Mean time to discover the next K faults → measure to predict the average length of time until the next K faults are encountered.

A "reliability tested product" will have the following characteristics.

- No errors or very few errors from repeated transactions
- Zero downtime
- Optimum utilization or resources
- Consistent performance and response time of the product
- No side-effects after the repeated transactions are executed.

3. Stress Testing

- Stress testing is done to ***evaluate a system beyond the limits of specified requirements or resources, to ensure that system does not break***
- Stress testing helps in understanding ***how the system can behave under extreme and realistic situations***
- It also helps to know the conditions under which these tests fail so that the maximum limits, in terms of simultaneous users, search criteria, large number of transactions, etc. can be known

In stress testing the load is generally increased through various means such as increasing the number of clients, users, and transactions till and beyond the resources is completely utilized. When the load keeps on increasing, the product reaches a ***stress point*** when some o the transactions start failing due to resources not being available. The failure rates may go up beyond this point. To continue the stress testing, the load is slightly reduced below this stress point to see whether the product recovers and whether the failure rate decreases appropriately.

Reasons for the product may not recover immediately when the load is decreased.

Some transactions may be in the wait queue, delaying the recovery

Some rejected transactions may need to be purges, delaying the recovery

Due to failure, some clean-up operations may be needed by the product, delaying the recovery

Certain data structures may have got corrupted and may permanently prevent recovery from stress point.

The time required for the product to quickly recover from those failures is represented by ***MTTR (Mean Time To Recovery)***

The following guidelines can be used to select the tests for stress testing.

- **Repetitive Tests**: Executing repeated tests ensures that at all times the code works as expected.
- **Concurrency**: Ensure that the code is exercised in multiple paths and simultaneously.
- **Magnitude**: Amount of load to be applied to the product to stress the system.
- **Random Variation**: Tests that stress the system with random inputs at random instances and random magnitude are selected and executed as part of stress testing.

4. Interoperability Testing

Interoperability testing is done ***to ensure the two or more products can exchange information, use information, and work properly together***.

Integration is a **method** and interoperability is the **end results**. Integration pertains to only one product and defines interfaces for two or more components. Unless two or more products are designed for exchanging information, interoperability cannot be achieved.

There are no real standard methodologies developed for interoperability testing. Following technical standards like SOAP (Simple Object Access Protocol), eXtensible Markup Language (XML) and some more from W3C (World Wide Web Consortium) typically aid in the development of products using common standards and methods.

Guidelines that help in improving interoperability

1. **Consistency of information flow across system**: When data structures are used to pass information across systems, the structure and interpretation of these data structures should be consistent across the system.
2. **Changes to data representation as per the system requirements**: When a little end-ian machine passes data to a big end-ian machine, the byte ordering would have to be changed.
3. **Correlated interchange of messages and receiving appropriate responses**: When one system sends an input in the form of a message, the next system is in the waiting mode or listening mode to receive the input.
4. **Communication and messages**: When a message is passed on from a system A to system B, if any and the messages is lost or gets grabbled the product should be tested to check how it responds to such erroneous messages.
5. **Meeting quality factors**: When 2 or more products are put together, there is an additional requirement of information exchange between them. This requirement should not take away the quality of the products that would have been already met individually by the products.

5. Usability Testing

Usability Testing is a type of testing done from an end-user's perspective to determine if the system is easily usable.

Usability Testing: Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions.

Systems may be built 100% in accordance with the specifications. Yet, they may be 'unusable' when it lands in the hands of the end-users. For instance, let's say a user needs to print a Financial Update Report, every 30 minutes, and he/she has to go through the following steps:

1. Login to the system
2. Click Reports
3. From the groups of reports, select Financial Reports
4. From the list of financial reports, select Financial Update Report
5. Specify the following parameters
    1. Date Range
    2. Time Zone
    3. Departments
    4. Units
6. Click Generate Report
7. Click Print
8. Select an option
    1. Print as PDF
    2. Print for Real

If that's the case, the system is probably practically unusable (though it functions perfectly fine). If the report is to be printed frequently, wouldn't it be convenient if the user could get the job done in a couple of clicks, rather than having to go through numerous steps like listed above? What if there was a feature to save frequently generated reports as a template and if the saved reports were readily available for printing from the homepage?

Usability Testing is normally performed during System Testing and Acceptance Testing levels.

Usability Testing is NOT to be confused with User Acceptance Testing or User Interface / Look and Feel Testing.

Usability Testing Checklist

Section I: Accessibility

→ Load time of Website is realistic.
→ Adequate Text-to-Background Contrast is present.
→ Font size & spacing between the texts is properly readable.
→ Website has its 404 page or any custom designed Not Found page.
→ Appropriate ALT tags are added for images.

Section II: Navigation

- Check if user is effortlessly recognizes the website navigation.
- Check if number of buttons/links are reasonable
- Check if the Company Logo Is Linked to Home-page
- Check if style of links is consistent on all pages & easy to understand.
- Check if site search is present on page & should be easy to accessible.

Section III: Content

- Check if URLs Are Meaningful & User-friendly
- Check if HTML Page Titles Are Explanatory
- Check if Emphasis (bold, etc.) Is Used Sparingly
- Check if Major Headings Are Clear & Descriptive
- Check if Styles & Colors Are Consistent

Key Benefits of Usability Testing:

- Decrease development and redesign cost which increases user satisfaction.
- User productivity increases, cost decreases.
- Increase business due to satisfied customers.
- Reduces user acclimation time and errors.
- Shorten the learning curve for new users.

Advantages of Usability Testing:

- Usability testing finds important bugs and potholes of the tested application which will be not visible to the developer.
- Using correct resources, usability test can assist in fixing all problems that user face before application releases.
- Usability test can be modified according to the requirement to support other types of testing such as functional testing, system integration testing, Unit testing, smoke testing.
- Planned *Usability testing* becomes very economic, highly successful and beneficial.
- Issues and potential problems are highlighted before the product is launched.

Limitations of usability testing:

- Planning and data-collecting process are time consuming.
- Always be confusing that why usability problems come.
- Its small and simple size makes it unreliable for drawing conclusions about subjective user preferences.
- It's hard to create the suitable context.
- You can't test long-term experiences.
- People act in a different way when they know they're being observed.

4.1 PEOPLE AND ORGANIZATIONAL ISSUES IS TESTING

COMMON PEOPLE ISSUES

4.1.1 Perceptions and Misconceptions about testing

- ***Testing is not technically challenging***
  - Requires a holistic understanding of the entire product
  - Requires thorough understanding of multiple domains
  - Specialization in languages, Use of tools
  - Opportunities for conceptualization and out-of-the-box thinking.
  - Significant investments are made in testing today – sometimes a lot more than in development
- ***Testing does not provide me a career path or growth***
  - Normally job titles are given as "Development Engineers", "Senior Development Engineer" etc.
  - Testing organizations do not always present such obvious opportunities for career growth. This does not mean that there are no career paths for testing professionals.
  - There is an equally lucrative career path for testing professionals also.
- ***I am put in testing – what is wrong with me?!***
  - Toppers allocated for development and testing functions get the leftovers, obviously management is sending the wrong signals and reinforcing the wrong message.
  - A person assigned testing only when he or she has the right aptitude and attitude for testing.
  - Compensations and reward favor the development leads to people "graduate to development" rather than look for careers in testing itself
- ***These folks are my adversaries***
  - Testing and development teams should reinforce each other and not be at loggerheads.
- ***Testing is what I can do in the end if I get time***
  - Testing is not what happens in the end of a project – it happens throughout and continues even beyond a release
- ***There is no sense of ownership in testing***
  - Testing has deliverables just as development has and hence testers should have the same sense of ownership
- ***Testing is only destructive***
  - Testing is destructive as much it is constructive, like the two sides of a coin.

Providing Career Path for Testing Professional

When people look for a career path in testing, some of the areas of progression they look for are,

- Technical challenge
- Learning opportunities
- Increasing responsibilities and authority
- Increasing independence
- Ability to have a significant influence
- Rewards and recognition

Responsibilities of a Test Engineer

- Following the test process for executing tests, maintaining tests etc.
- Filing high quality defects, usable by developers
- Categorizing defects
- Adhering to schedules specified
- Developing high quality documentation

Responsibilities of a Senior Test Engineer

- Helping development staff in debugging and problem isolation
- Contribution to enhancing processes for testing
- Generation of metrics related to testing

Responsibilities of a Test Lead

- Review of test case, test design etc.
- Planning test strategy
- Allocating tasks to individual and monitoring it
- Mentoring team members and assisting them in technical matters
- Interaction with developing team for debugging and problem reproduction
- Overall responsibility for test quality

Responsibilities of a Test Architect

- A test architect has in-depth knowledge of a variety of testing techniques and methodologies used both inside and outside of an organization
- They often provide technical assistance and/or advice to the test Manager.
- Test Architect come into picture when Test Manager takes on additional responsibilities
- A test architect is expected to be able to affect change not only across the testing community, but between other engineering disciplines as well.

4.2.1 Development Functions Vs Testing

Similarities

1. **Requirement / Test Specification** : Requires thorough understanding of the domain.
2. **Design** : Carries with it all attributes for product design like reuse, standard formation.
3. **Coding / Test Script** : Involves using the development and test automation tools.
4. **Testing / Making the tests operational** : Involves well-knot teamwork between teams to ensure that correct results are captured.
5. **Maintenance** : Keeping the tests current with changes from maintenance.

Differences

1. Testing is often a crunch time function
   Testing functions close to product release time throws in some unique planning and management challenge.
2. More "elasticity" is allowed in projects in earlier phases
   Development function will take longer than planned, whereas same amount is not given for testing as final deadline for a product release is seldom compromised.
3. Testing is arguably the most difficult ones to staff
   It is difficult to attract and retain top talent for testing functions.
4. Testing usually carry more external dependencies that development functions.

The role of the ecosystem and call for action

- ✓ ***Role of education system***: The right values can only be more effectively **caught** by the students than be **taught** by the teachers.
  - Not been as core course; No lab experience; No real time experience
  - Projects not asking test plan, but for coding only
  - Scope for team work???
- ✓ ***Role of senior management***: Fairness to and recognition of testing professionals should not only be **done** but should be **seen to** be done.
  - How to spot a good tester.
  - Rewarding people.
- ✓ ***Role of the community***: As members of test community, do you have pride and sense of equality/ Remember, authority is **taken, not given**.
  - Pride in work

4.2 ORGANISATION STRUCTURES FOR TESTING TEAMS

Dimensions of Organization Structure

- **Organization Type**
- **Geographic Distribution**

  - Organization types
    - **Product**
      - Responsibility for entire product
      - Testing is one phase
    - **Service**
      - Provide testing service to other organizations
      - Provide test specialist
  - Geographic distribution
    - Single site → all members at one place
    - Multi site → members at different location

4.2.1 *Testing Team Structure for Single-Product Companies*

- Most product companies start with a single product.
- During the initial stages of evolution, the organization does not work with many formalized processes.
- The product delivery team members distribute their time among multiple tasks and after wear multiple hats.

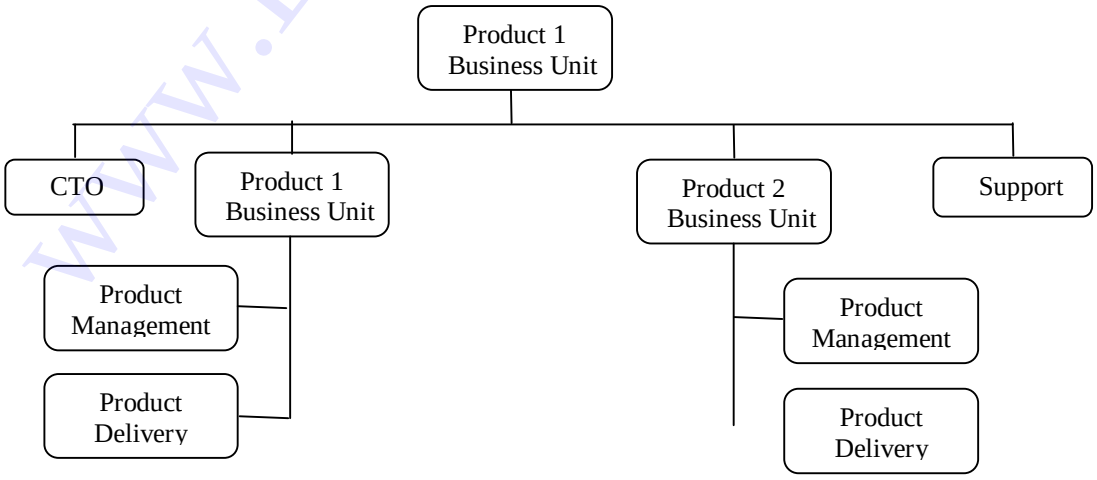
- **Advantages**
  - ❖ Exploits the rear-loading nature of testing activities (during early part of the project, everyone chips in for development and during the later part, they switch over to test)
  - ❖ Enables engineers to gain experience in all aspects of life cycle.
  - ❖ Is amenable to the fact that the organization mostly only has informal processes.
  - ❖ Some defects may be detected early.

- **Disadvantages**
  - ❖ Accountability for testing and quality reduces (give importance for developing or testing?)
  - ❖ Developers do not in general like testing and hence the effectiveness of testing suffers
  - ❖ Schedule pressures generally compromise testing (deadline to meet leads to insufficient time for testing causes compromise in quality of testing).
  - ❖ Developers may not be able carry out the different types of tests.

4.2.2 *Testing Team Structure for Multi-Product Companies*

The organization of test teams in multi-product companies is dictated largely by the following factors.

- How tightly couples the products are in terms of technology
- Dependence among various products
- How synchronous are the release cycles of products
- Customer base for each product & similarity among customer bases for various products.



Based on the above factors, there are several options available for organizing testing teams.

1. Testing team as part of "CTO's Office"
   - Developing a product architecture that is testable or suitable for testing
   - Testing team will have better product and technology skills
   - The testing team can get a clear understating of what design and architecture are built for and plan their tests accordingly
   - The CTO's team can evolve a consistent, cost-effective strategy for test automation.
   - In order to make the test team more effective,
     - It should be smaller in number
     - It should be a team of equals or at most very few hierarchies.
     - It should have organization-wide representation
     - It should have decision-making and enforcing authority
     - It should be involved in periodic reviews to ensure operations are in line.
2. Single test team for all products.

   This model is similar to the case of a single product team divided into multiple components and each of the components being developed by an independent team. Here, since different groups have delivery responsibilities for different products, the single testing team must report to a different level. The possibilities are,

   (i) The single testing team can form a "testing business unit" and report into this unit.

   (ii) The testing team can be made to report to the "CTO thing-tank".
3. Testing teams organized by product

   The issues in single testing teams are, accountability, decision making and scheduling. Solution?...

   Assign complete responsibility of all aspects of a product to the corresponding business unit and let the business unit head figure out how to organize the testing and development teams.
4. Separate testing team for different phases of testing

Types of test	Reports into
White box testing	Development team
Black box testing	Testing team
Integration testing	Organization-wide testing team
System testing	Product management / Product marketing
Performance testing	A central benchmarking group
Acceptance testing	Product management / Product marketing
Internationalization testing	internationalization team and some local teams
Regression testing	All test Teams

Advantages

1. People with appropriate skill sets are used to perform a given type of test.
2. Defects can get detected better and closer to the point of injection.

4.3 TESTING SERVICES – (SERVICE STRUCTURE OF TESTING TEAM)

4.3.1 Testing Services Organization

Today it is common to find testing activities outsourced to external companies who specialize is testing and provide testing services.

Business Need for Testing Services

1. Testing is becoming increasingly diverse and a very specialized function
2. The variety and complexity of the test automation tools further increase the challenge in testing.
3. Testing as a process is becoming better defined and thus makes testing to outsourcing.
4. An organization expertise in understanding software domain may be expert in setting up and running an effective testing function.
5. An outsourced organization can offer location and cost advantage.

Typical Roles and Responsibilities of Testing Services Organization

- A testing services organization is made up of a number of accounts.
- Each account being responsible for a major customer.
- Each account is assigned an account manager.
- ***Account manager***, a single point of contact from the customer into testing services organization.

  Account manager service:
  - Single point contact between customer and testing service organization.
  - Develops rapport with customer; responsible for ensuring current projects are delivered as promised; getting new business.
  - Participate in all strategic (and tactical) communication between them.
  - Acts as a proxy for the customer within the testing services organization.

- ✓ Account manager may be ***located*** close to the customer site or at the location of testing services organization.
- ✓ To develop better rapport, this role would ***require frequent travel and face-to-face meetings*** with the customer.

The testing service team organizes its account team as a ***near-site team*** and a ***remote team***.

Near-site team → Usually a small team, placed at or near customer location.

- Direct first point of contact for the costumer for tactical and urgent issues
- Act as a stop-gap to represent the remote team, in the event of emergencies.
- Serves to increase the rapport between the customer's operational team and the testing services team.

Remote team → Usually large in number the team does the bulk of work, located on the site of the testing services organization.

- The ***remote team manager*** manages the entire remote team.
- Can have a peer-to-peer relationship with the near-site team or have the near-site team reporting to them.
- Can have further hierarchies as test leads and test engineers.

Challenges and Issues in Testing Service Organizations

1. The outsider effect and estimation of resources
   - Do not necessarily have access to the product internals or code.
   - Do not have access to product history (to find which modules has historically problem prone – results been in not able to prioritize testing).
   - May not necessarily have same level of rapport with development team.
   - Will have to estimate and plan for hardware and software resources.
2. Domain expertise
   - Testing service organization may have to undertake projects from multiple customers.
   - The little product ownership makes it tougher to get domain expertise to join a testing service company.
   - The diversity of domains exacerbates the problem.
3. Privacy and customer isolation issues
   - As testing service organization has a common infrastructure, physical isolation of the different teams may be difficult
   - As people more from one project to another, there should be full confidence and transparency in ensuring the customer-specific knowledge acquired in one project is not taken to other projects.
   - A Non Disclosure Agreement (NDA) is draw up between the customer and the testing service organization.
4. Apportioning hardware and software resources and costs
   - When dealing with multiple customers, the organization uses hardware and software resources internally.
   - Some of them are identified and allocated for particular account and some others and multiplexed across multiple projects (eg. Satellite links, e-mail server etc.).
   - The cost has to be apportioned across different projects while costing the project.
5. Maintaining the “bench”
   - Need to maintain “people of bench” – people not allocated to any project but ready in the wings to take on new projects and to convince customers.
     - New projects may come at any time
     - Some prospects may require initial studies to be done or some demonstration of initial capability before signing.
     - May need to know resumes of specific individuals and to select people at initial level.

Success Factors for Testing Organizations

(i) Communication and teamwork
(ii) Bringing in customer perspective
(iii) Providing appropriate tools and environment
(iv) Providing periodic skill upgrades

4.4 TEST PLANNING

A plan is a document that provides a framework or approach for achieving a set of goals.

In order to meet a set of goals, a plan describes what specific tasks must be accomplished, who is responsible for each task, what tools, procedures, and techniques must be used, how much time and effort is needed, and what resources are essential. A plan also contains milestones.

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.

Tracking the actual occurrence of the milestone events allows a manager to determine if the project is progressing as planned. Finally, a plan should assess the risks involved in carrying out the project.

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.

Test plans for software projects are very complex and detailed documents. The planner usually includes the following essential high-level items.

1. *Overall test objectives*. As testers, why are we testing, what is to be achieved by the tests, and what are the risks associated with testing this product?

2. *What to test* *(scope of the tests)*. What items, features, procedures, functions, objects, clusters, and subsystems will be tested?
3. *Who will test*. Who are the personnel responsible for the tests?
4. *How to test*. What strategies, methods, hardware, software tools, and techniques are going to be applied? What test documents and deliverable should be produced?
5. *When to test*. What are the schedules for tests? What items need to be available?
6. *When to stop testing*. It is not economically feasible or practical to plan to test until all defects have been revealed. This is a goal that testers can never be sure they have reached. Because of budgets, scheduling, customer deadlines, specific conditions must be outlined in the test plan.

Test plans can be organized in several ways depending on organizational policy. The complexity of the hierarchy depends on the type, size, risk-proneness, and the mission/safety criticality of software system being developed. All of the quality and testing plans should also be coordinated with the overall software project plan. A sample plan hierarchy is shown in following figure.

Software quality assurance plan → This plan gives an overview of all verification and validation activities for the project, details related to other quality issues such as audits, standards, configuration control, and supplier control.

Master test plan → An overall description of all execution-based testing for the software system.
Master verification plan → For reviews inspections/walkthroughs The master test plan itself may be a component of the overall project plan or exist as a separate test plan for unit, integration, system, and acceptance tests.
The level-based plans give a more detailed view of testing appropriate to that level.

The **persons responsible** for developing test plans depend on the type of plan under development. For example, the **master test plan** for execution-based testing may be developed by the **project manager**, especially if there is no separate testing group. A **tester or software quality assurance manager**, can also do this but always requires cooperation and input from the project manager.

The type and organization of the test plan, the test plan hierarchy, and who is responsible for development should be specified in organizational standards or SQA documents.

4.5 TEST PLAN COMPONENTS

The basic test plan components as described in IEEE Std 829-1983 is shown in following figure.

1. **Test Plan Identifier**

- Each test plan should have a unique identifier so that it can be associated with a specific project and become a part of the project history.
- The project history and all project-related items should be stored in a project database
- Organizational standards should describe the format for the test plan identifier and how to specify versions

2. Introduction

- Here, the test planner gives an overall description of the project, the software system being developed or maintained, and the software items and/or features to be tested.
- References to related or supporting documents should also be included
- If test plans are developed as multilevel documents, then each plan must reference the next higher level plan for consistency and compatibility reasons.

3. Items to Be Tested

- This is a listing of the entities to be tested and should include names, identifiers, and version/revision numbers for each entity.
- The items listed could include procedures, classes, modules, libraries, subsystems, and systems.
- References to the appropriate documents and the user manual should be included
- The test planner should also include items that will *not* be included in the test effort.

4. Features to Be Tested

- ***Features may be described as distinguishing characteristics of a software component or system.***
- Features that will *not* be tested should be identified and reasons for their exclusion from test should be included.
- References to test design specifications for each feature and each combination of features are identified

5. Approach

- Provides broad coverage of the issues to be addressed when testing the target software.
- Testing activities are described.
- Tools and techniques necessary for the tests should be included.
- Expectations for test completeness and how the degree of completeness will be determined should be described

6. Item Pass/Fail Criteria

- Given a test item and a test case, the tester must have a set of criteria to decide on whether the test has been passed or failed upon execution.
- A failure occurs when the actual output produced by the software does not agree with what was expected, under the conditions specified by the test.
- Scales are used to rate failures/defects with respect to their impact on the customer/user

7. Suspension and Resumption Criteria

- In the simplest of cases testing is suspended at the end of a working day and resumed the following morning.
- The test plan should also specify conditions to suspend testing based on the effects or criticality level of the failures/defects observed.
- Conditions for resuming the test after there has been a suspension should also be specified.

8. Test Deliverables

- Test cases describe the actual test inputs and expected outputs.
- Deliverables may also include other documents that result from testing such as test logs, test transmittal reports, test incident reports, and a test summary report.
- Another test deliverable is the test harness that is supplementary code that is written specifically to support the test efforts
- Support code, like, testing tools that will be developed especially for this project, should also be described

9. Testing Tasks

- Identify all testing-related tasks and their dependencies using a Work Breakdown Structure (WBS)
- ***A Work Breakdown Structure is a hierarchical or treelike representation of all the tasks that are required to complete a project.***
- High-level tasks sit at the top of the hierarchical task tree.
- Leaves are detailed tasks sometimes called work packages that can be done by 1–2 people in a short time period, typically 3–5 days.

10. The Testing Environment

- Here the test planner describes the software and hardware needs for the testing effort (Eg.) emulators, telecommunication equipment, etc.
- The planner must also indicate any laboratory space containing the equipment that needs to be reserved.
- The planner also needs to specify any special software needs such as coverage tools, databases, and test data generators.

11. Responsibilities

- The staff who will be responsible for test-related tasks should be identified.
- This includes personnel like, developers, testers, software quality assurance staff, systems analysts, and customers/users, who will be:
  - ✓ developing test design specifications, and test cases;
  - ✓ executing the tests and recording results;
  - ✓ checking results;
  - ✓ interacting with developers;
  - ✓ developing the test harnesses;
  - ✓ interacting with the users/customers.

12. Staffing and Training Needs

- The test planner should describe the staff and the skill levels needed to carry out test-related responsibilities
- Any special training required to perform a task should be noted.

13. Scheduling

- Task durations should be established and recorded with the aid of a task networking tool.
- Test milestones should be established, recorded, and scheduled.

- Schedules for use of staff, tools, equipment, and laboratory space should also be specified.

14. Risks and Contingencies

- Every testing effort has risks associated with it.
- Testing software with a high degree of criticality, complexity, or a tight delivery deadline all impose risks that may have negative impacts on project goals.
- These risks should be: (i) identified, (ii) evaluated in terms of their probability of occurrence, (iii) prioritized, and (iv) contingency plans should be developed that can be activated if the risk occurs.

15. Testing Costs

- The project manager in consultation with developers and testers estimates testing costs.
- If the test plan is an independent document prepared by the testing group and has a cost component, the test planner will need tools and techniques to help estimate test costs.
- Test costs that should included in the plan are:
  - ✓ costs of planning and designing the tests;
  - ✓ costs of acquiring the hardware and software necessary for the tests;
  - ✓ costs to support the test environment;
  - ✓ costs of executing the tests;
  - ✓ costs of recording and analyzing test results;
  - ✓ tear-down costs to restore the environment.

16. Approvals

- The test plan(s) for a project should be reviewed by those designated by the organization.
- All parties who review the plan and approve it should sign the document.

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

4.6 TEST PLAN ATTACHMENTS

4.6.1 Test Design Specifications

- The IEEE standard for software test documentation describes a test design specification as a test deliverable that specifies the requirements of the test approach.
- The test design specification also has links to the associated test cases and test procedures needed to test the features, and also describes in detail pass/fail criteria for the features.
- To develop test design specifications many documents such as the requirements, design documents, and user manual are useful.
- A test design specification should have the following components according to the IEEE standard.

Test Design Specification Identifier → Give each test design specification a unique identifier and a reference to its associated test plan.

Features to Be Tested → Test items, features, and combination of features covered by this test design specification are listed.

Approach Refinements → In the test plan a general description of the approach to be used to test each item was described.
The test planner also describes how test results will be analyzed.
The relationships among the associated test cases are discussed.

Test Case Identification → Each test design specification is associated with a set of test cases and a set of set procedures.
The test cases contain input/output information
Test procedures contain the steps necessary to execute the tests.

Pass/Fail Criteria → The specific criteria to be used for determining whether the item has passed/failed a test.

4.6.2 Test Case Specifications

- This series of documents attached to the test plan defines the test cases required to execute the test items named in the associated test design specification.
- Each test case must be specified correctly so that time is not wasted in analyzing the results of an erroneous test.

Test Case Specification Identifier → Each test case specification should be assigned a unique identifier.

Test Items → This component names the test items and features to be tested by this test case specification.

Input Specifications → This component of the test design specification contains the actual inputs needed to execute the test.

Output Specifications → All outputs expected from the test should be identified. If an output is to be a specific value or a specific feature it also should be stated. The output specifications are necessary to determine whether the item has passed/failed the test.

Special Environmental Needs → Any specific hardware and specific hardware configurations needed to execute this test case should be identified. Special software required executing the test such as compilers, simulators, and test coverage tools should be described.

Special Procedural Requirements → Describe any special conditions or constraints that apply to the test procedures associated with this test.

Intercase Dependencies → The test planner should describe any relationships between this test case and others, and the nature of the relationship.

4.6.3 Test Procedure Specifications

A procedure in general is a sequence of steps required to carry out a specific task.

- The planner specifies the steps required to execute a set of test cases.
- It specifies the steps necessary to analyze a software item in order to evaluate a set of features.

Test Procedure Specification Identifier → Each test procedure specification should be assigned a unique identifier.

Purpose → Describe the purpose of this test procedure and reference any test cases it executes.

Specific Requirements → List any special requirements for this procedure, like software, hardware, and special training.

Procedure Steps → Here the actual steps including methods, documents for recording (logging) results, and recording incidents are described.

- Steps include:
  - Setup : to prepare for execution of the procedure;
  - Start : to begin execution of the procedure;
  - Proceed : to continue the execution of the procedure;
  - Measure : to describe how test measurements related to outputs will be made;
  - Shut down : to describe actions needed to suspend the test when unexpected events occur;
  - Restart : to describe restart points and actions needed to restart the procedure from these points;
  - Stop : to describe actions needed to bring the procedure to an orderly halt;

4.7 LOCATING TEST ITEMS

4.7.1 The Test Item Transmittal Report

Suppose a tester is ready to run tests on an item on the date described in the test plan. She needs to be able to locate the item and have knowledge of its current status. This is the ***function of the Test Item Transmittal Report***.

This document is not a component of the test plan, but is necessary to locate and track the items that are submitted for test. Each Test Item Transmittal Report has a unique identifier. It should contain the following information for each item that is tracked.

(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.

4.8 TEST MANAGEMENT

Test managements includes aspects that should be taken care of in planning a project. These aspects are proactive measures that can have an across-the-board influence in all testing projects.

4.8.1 Choice of Standards

Standards are of 2 types, external and internal standards. External standards are standards that a product should comply with, are externally visible, and are usually stipulated by external consortia. Internal standards are standards formulated by a testing organization to bring in consistency and predictability. Some of the internal standards include,

(i) Naming and storage conventions for test artifacts

- Every test artifact should be named appropriately and meaningfully.
- Stipulates the conventions for directory structure for tests.

(ii) Document standards

- For manual testing, documentation standards correspond to specifying the user and system responses at the right level of detail that is consisten with the skill level of the tester.

(iii)Test coding standards

- Test coding standards go one level deeper into the tests and enforce standards on how the tests themselves are written.

(iv)Test reporting standards

- The stakeholders must get a consistent and timely view of the progress of tests.
- It provides guidelines on the level of detail that should be present in the test reports.

4.8.2 *Test Infrastructure Management*

Testing requires a robust infrastructure to be planned upfront. This infrastructure is made up of 3 essential elements.

- A test case database (TCDB) → captures all the relevant information aboult the test cases in an organization.
- A defect repository → captures all the relevant details of defects reported for a product. It is an important vehicle of communication that influences the work flow ithin a software organization.
- Configuration management repository and tool → keeps trace of change control and version control of all the files / entities that make up a software product.

4.8.3 *Test People Management*

- People management is an integral part of any project management.
- A person relies only on his or her own skills to accomplish an assigned activity
- It requires the ability to taught (unlike technical skills).
- Success of a testing organization depends vitally on judicious people managemtn skills.
- The common goals and the spirit of teamwork have to be internalized by all the stakeholders.

4.8.4 *Integrating with Product Release*

- The success of a product depends on the effectiveness of integration of the development and testing activities.
- Project planning for the entire product should be done in a holistic way.
- Some of the points to be decided for this planning are as follows.
    - Sync points between development and testing as to when different types of testing can commence. (Eg. When integration testing could start? When system testing could start?)
    - Services level agreements between development and testing as to how long it would take for the testing team to complete the testing.
    - Consistent definitions of the various properties and severities of the defects.
    - Communication mechanism to the documentation group to ensure that the documentation is kept in sync with the product.

4.9 TEST PROCESS

- **Putting Together and Baselining a Test Plan**
    - An organization develops a template that is to be used across the board and each testing project puts together a test plan based on it.
    - A change, if any, is made only after careful deliberations (proper approval).

- The test plan is reviewed by a designated set of competent people of the organization.
- It is approved by a competent authority, an independent of project manager directly responsible for testing.
- The test plan is base lined into the configuration management repository.
- Now, the base lined test plan becomes the bases for running the testing project.

- **Test case Specification**
  - **Test case specification is designed by the testing team based on test plan.**
  - **It becomes the bases for preparing individual test cases.**
  - A test case specification should clearly identify,
    - The purpose of test
    - Items being testing along with their version or release number
    - Environment that needs to be set up for running the test case.
    - Input data to be used for the test case.
    - Steps to be followed to execute the test.
    - The expected results that are considered to be "correct results".
    - A step to compare the actual results produced with the expected results.
- Update of Traceability Matrix
  - The traceability matrix is a tool to validate that every requirement is tested.
  - It is created during the requirements gathering phase itself by filling up the unique identifier for each requirement.
  - On completion, the row corresponding to the requirement which is being tested by the test case is updated with the test case specification identifier.
- Identifying Possible Candidates for Automation.
  - Before writing test cases, decision should be taken as to which tests are to be automated and which should run manually.
  - Some criteria that will be used in deciding for automate include,
    - Repetitive nature of the test
    - Effort involved in automation
    - Amount of manual intervention required for the test
    - Cost of automation tool
- Developing and Baselining Test Cases
  - Test case development entails translating the test specifications to a form from which the tests can be executed.
    - For automation, it requires writing test scripts in the automation language.
    - For manual, it maps to writing detailed step-by-step instructions for executing the tests and validating the results.
    - Test case should have change history documents, which specifies,
      - What was the change
      - Why the change was necessitated
      - Who made the change
      - When was the change made
      - Brief description of how the change has been implemented
      - Other files affected by the change

- Collecting and Analyzing Metrics
  - Information about test execution gets collected in test logs and other files.
- Preparing Test Summary Report
  - On test cycle completion, a test summary report is produced.
  - It gives insights to the senior management about the fitness of the product for release.
- Recommending Product Release Criteria
  - Defect identification is an evidence of what defects exist in the product, their severity and impact.
    - What defects the product has
    - What is the impact / severity of each of the defects
    - What would be the risks of releasing the product with the existing defects?

4.10 REPORTING TEST RESULTS

The test plan and its attachments are test-related documents that are prepared *prior* to test execution. There are additional documents related to testing that are prepared during and after execution of the tests. The *IEEE Standard for Software Test Documentation* describes the following documents.

4.10.1 Test Log

The test log, **a diary of events during tests**, should be prepared by the person executing the tests. In the experimental world of engineers and scientists detailed logs are kept when carrying out experimental work.

The test log is invaluable for use in defect repair. It gives the developer a snapshot of the events associated with a failure. The combination of test log and test incident documents helps to prevent incorrect decisions based on incomplete or erroneous test results that often lead to repeated, but ineffective, test-patch-test cycles.

The test log is valuable for (i) regression testing that takes place in the development of future releases of a software product, and (ii) circumstances where code from a reuse library is to be reused.

The test log can have many formats. An organization can design its own format or adopt IEEE recommendations. The IEEE Standard for Software Test Documentation has the following sections:

1. *Test Log Identifier:* Each test log should have a unique identifier.
2. *Description:* Tester should identify the items being tested, their version/revision number, and their associated Test Item/Transmittal Report. The environment in which the test is conducted should be described including hardware and operating system details.

3. *Activity and Event Entries:* The tester should provide dates and names of test log authors for each event and activity. This section should also contain:

- *Execution description:* Provide a test procedure identifier and also the names and functions of personnel involved in the test.
- *Procedure results:* For each execution, record the results and the location of the output.
- *Incident report identifiers:* Record the identifiers of incident reports generated while the test is being executed.

4.10.2 Test Incident Report

The tester should record in a test incident report (sometimes called a **problem report**) any event that occurs during the execution of the tests that is ***unexpected, unexplainable, and that requires a follow-up investigation***.

The *IEEE Standard for Software Test Documentation* recommends the following sections in the report:

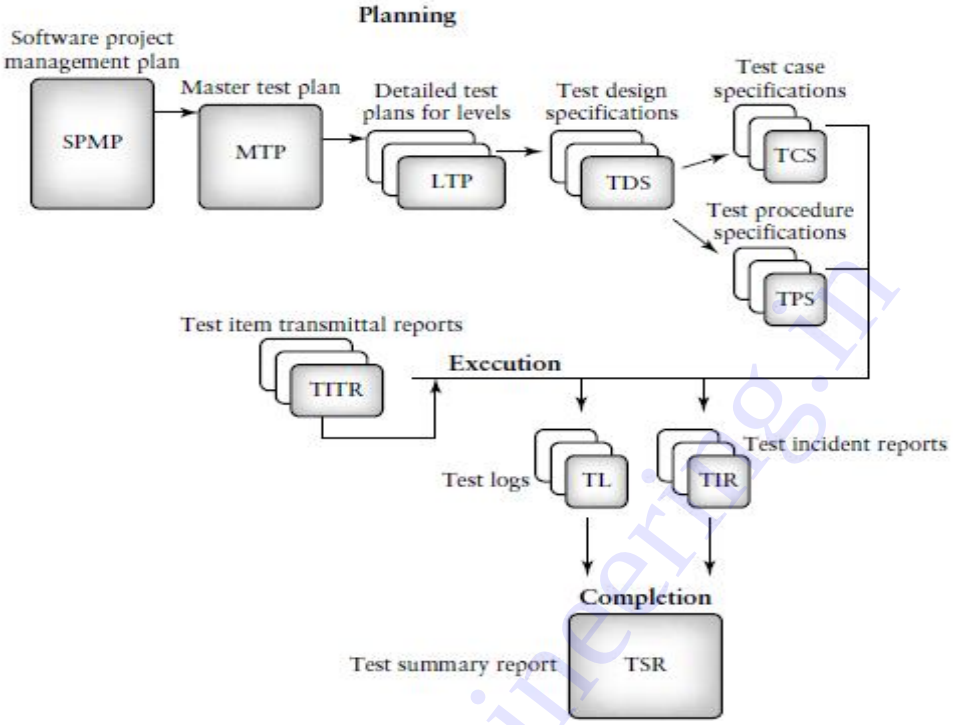
1. *Test Incident Report identifier:* to uniquely identify this report.
2. *Summary:* to identify the test items involved, the test procedures, test cases, and test log associated with this report.
3. *Incident description:* to describe time and date, testers, observers, environment, inputs, expected outputs, actual outputs, procedure step etc.
4. *Impact:* what impact will this incident have on the testing effort, the test plans, the test procedures, and the test cases? (severity rating)

4.10.3 Test Summary Report

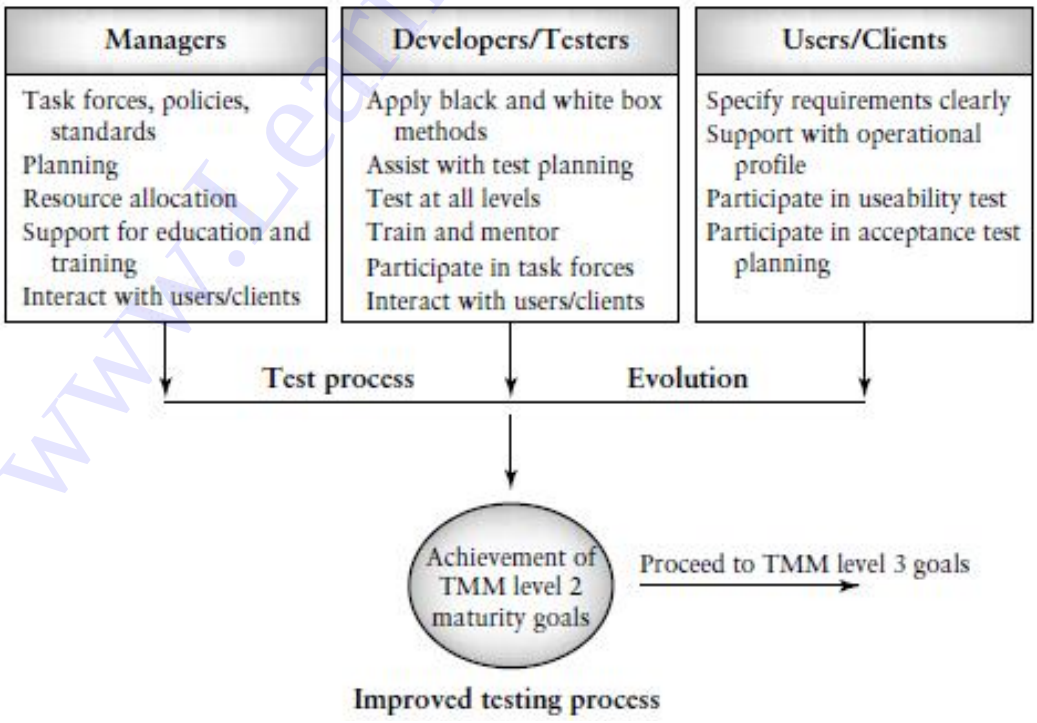
This report is prepared when testing is complete, a summary of the results of the testing efforts. When a project postmortem is conducted, the Test Summary Report can help managers, testers, developers, and SQA staff to evaluate the effectiveness of the testing efforts. The IEEE test documentation standard describes the following sections for the Test Summary Report:

1. *Test Summary Report identifier:* to uniquely identify this report.
2. *Variances:* variances of the test items from their original design. Deviations and reasons for the deviation from the test plan, test procedures, and test designs are discussed.
3. *Comprehensiveness assessment:* the document author discusses the comprehensiveness of the test effort as compared to test objectives and test completeness criteria as described in the test plan.
4. *Summary of results:* the document author summarizes the testing results. Resolved and unresolved incidents should be described.
5. *Evaluation:* author evaluates each test item based on test results. Did it pass/fail the tests? If it failed, what was the level of severity of the failure?
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.

The following figure shows the relationships between all the test-related documents.



4.11 THE ROLE OF THE THREE GROUPS IN TESTING PLANNING AND POLICY DEVELOPMENT



The **manager's view** involves commitment and support for those activities and tasks related to improving testing process quality.
The **developer/tester's view** encompasses the technical activities and tasks that when applied, constitute best testing practices.
The **user/client view** is defined as a cooperating or supporting view.

Critical group participation is summarized in following figure.

4.12 INTRODUCING THE TEST SPECIALIST

The organization tests its software at several levels (unit, integration, system, etc.) Moving up to next level requires further investment of organizational resources in the testing process. One of the maturity goals at this level calls for the "Establishment of a test organization." It implies a commitment to better testing and higher-quality software. This commitment requires that testing specialists be hired, space be given to house the testing group, resources be allocated to the group, and career paths for testers be established.

Although there are many costs to establishing a testing group, there are also many benefits. By supporting a test group an organization acquires *leadership* in areas that relate to testing and quality issues. For example, there will be staff with the necessary skills and motivation to be responsible for:

- 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.

The staff members of such a group are called test specialists or test engineers. Their primary responsibly is to ensure that testing is effective and productive, and that quality issues are addressed.

Testers are not developers, or analysts, although background in these areas is very helpful and necessary. Testers don't repair code. However, they add value to a software product in terms of higher quality and customer satisfaction. They are not destructive; they are constructive.

Test specialists need to be educated and trained in testing and quality issues.

4.13 SKILLS NEEDED BY A TEST SPECIALIST

Given the nature of assigned to the tester, many managerial and personal skills are necessary for success in the area of work. On the *personal* and *managerial* level a test specialist must have various skill and all of these skills are summarized as follows.

Test specialist skills

Personal and Managerial Skills
Organizational, and planning skills
Track and pay attention to detail
Determination to discover and solve problems
Work with others, resolve conflicts
Mentor and train others
Work with users/clients
Written/oral communication skills
Think creatively

Technical Skills
General software engineering principles and practices
Understanding of testing principles and practices
Understanding of basic testing strategies, and methods
Ability to plan, design, and execute test cases
Knowledge of process issues
Knowledge of networks, databases, and operating systems
Knowledge of configuration management
Knowledge of test-related documents
Ability to define, collect, and analyze test measurements
Ability, training, and motivation to work with testing tools
Knowledge of quality issues

4.14 BUILDING A TESTING GROUP

Organizing, staffing, and directing were major activities required to manage a project and a process. Hiring staff for the testing group, organizing the testing staff members into teams, motivating the team members, and integrating the team into the overall organizational structure are organizing, staffing, and directing activities your organization will need to perform to build a managed testing process.

Establishing a specialized testing group is a major decision for an organization. The steps in the process are summarized in following figure.

The following gives a brief description of the duties for each tester that is common to most organizations.

4.14.1 The Test Manager

In organizations with a testing function, the test manager (or test director) is the central person concerned with all aspects of testing and quality issues. The test manager is usually responsible for,

- test policy making,
- customer interaction,
- test planning,
- test documentation,
- controlling and monitoring of tests,
- training,
- test tool acquisition,
- participation in inspections and walkthroughs,
- reviewing test work,
- the test repository,
- and staffing issues such as hiring, firing, and evaluation of the test team members.

4.14.2 The Test Lead

The test lead assists the test manager and works with a team of test engineers on individual projects. He or she may be responsible for duties such as,

(v) test planning, (ii) staff supervision, and (iii) status reporting.

The test lead also participates in test design, test execution and reporting, technical reviews, customer interaction, and tool training.

4.14.3 The Test Engineer

The test engineers design, develop, and execute tests, develop test harnesses, and set up test laboratories and environments. They also give input to test planning and support maintenance of the test and defect repositories.

4.14.4 The Junior Test Engineer

The junior test engineers are usually new hires. They gain experience by participating in test design, test execution, and test harness development. They may also be asked to review user manuals and user help facilities defect and maintain the test and defect repositories.

5.1 SOFTWARE TEST AUTOMATION

Developing software to test the software is called test automation.

Test automation advantages

1. Automation saves time as software can execute test cases faster than human do. The time thus saved can be sued effectively for test engineers to,
    a. Develop additional test cases to achieve better coverage
    b. Perform some esoteric or specialized tests like ad hoc testing; or
    c. Perform some extra manual testing.
2. Test automation can free the test engineers from mundane tasks and make them focus on more creative tasks.
3. Automated tests can be more reliable.
4. Automation helps is immediate testing.
5. Automation can protect an organization against attrition of test engineers
6. Test automation opens up opportunities for better utilization of global resources.
7. Certain types of testing cannot be executed without automation.
8. Automation means end-to-end, not test execution alone.

5.2 SKILLS NEEDED FOR AUTOMATION

Automation – First generation	Automation – Second generation	Automation – Third generation	
Skills for test case automation	Skills for test case automation	Skills for test case automation	Skills for framework
Record-playback tools usage	Scripting languages	Scripting languages	Programming languages
	Programming languages	Programming languages	Design and architecture skills for framework creation
	Knowledge of data generation technique	Design and architecture of the product under test	Generic test requirements for multiple products
	Usage of the product under test	Usage of the framework	

5.3 SCOPE FOR AUTOMATION

1. Identifying the Types of Testing Amenable to Automation
    a. Stress, reliability, scalability and performance testing.
    b. Regression tests
    c. Functional tests
2. Automating Areas Less Prone to Change
3. Automate Tests that Pertain to Standards
4. Management Aspects in Automation.

5.4 DESIGN AND ARCHITETURE FOR AUTOMATION

Fig. 16.2 in Text book.(Srinivasan and Gopalasamy)

- External Modules
    - TCDB and DB
    - The steps to execute all the test cases and the history of their execution is stored in TCDB
    - DB (Defect database / Defect Repository) contains details of all the defects that are found in various products that are tested in a particular organization.
- Scenario and Configuration File Modules
    - Scenarios are information on “how to execute a particular test case”.
    - A configuration file contains a set of variables that are used in automation. This file is important for running the test cases for various execution conditions and for running the tests for various input and output conditions and states.
- Test Cases and Test Framework Modules
    - A test case means that automated test cases that are taken form TCDB and executed by the framework.
    - A test framework is a module that combines “what to execute” and “how they have to be executed”.
    - The test framework is considered the core of automation design. It subjects the test cases to different scenarios.
    - The framework monitors the results of every iteration and the results are stored.
    - It can be developed by the organization internally or can be brought from the vendor.
- Tools and Results Modules
    - When a test framework performs its operations, there are set of tools that may be required
    - To run the compiled code, certain runtime tools and utilities may be required.
    - The results for each of the test case along with scenarios and variable values have to be stored for future analysis and action.
    - The history of all the previous tests run should be recorded and kept as archives.
- Report Generator and Reports / Metrics Modules
    - Preparing reports is a complex and time consuming effort and hence it should be part of automation design.

- There should be,
  - Customized reports such as an executive reports, which gives very high level status.
  - Technical reports, which give a moderate level of details of the tests run
  - Detailed or debug reports, which are generated for developers to debug the failed test cases and the product.
  - The module that takes the necessary inputs and prepares a formatted report is called ***a report generator***.
  - Once the results are available, the report generator can generate ***metrics***.

5.5 REQUIREMENTS FOR TEST TOOL / FRAMEWORK

1. No hard coding in the test suite
2. Test case / suite expandability
   a. Adding a test case should not affect other test cases
   b. Adding a test case should not result in retesting the complete test suite
   c. Adding a new test suite to the framework should not affect existing test suites.
3. Reuse of code for different types of testing, test cases
   a. The test suite should only do what a test is expected to do. The test framework needs to take care of "how", and
   b. The test programs need to be modular to encourage reuse of code.
4. Automatic setup and cleanup
5. Independent test cases
6. Test case dependency
7. Insulating test cases during execution
8. Coding standards and directory structure.
9. Selective execution of test cases
10. Random execution of test cases
11. Parallel execution of test cases
12. Looping the test cases
13. Grouping of test scenarios
14. Test case execution based on previous results
15. Remote execution of test cases
16. Automatic archival of test data
17. Reporting scheme
18. Independent of languages

5.6 TEST METRICS AND MEASUREMENTS

Metrics derive information from raw data with a view to help in decision making. Some of the areas are,

1. Relationship between the data points.
2. Any cause and effect correlation between the observed data points.

3. Any pointers to how the data can be used for future planning and continuous improvements.

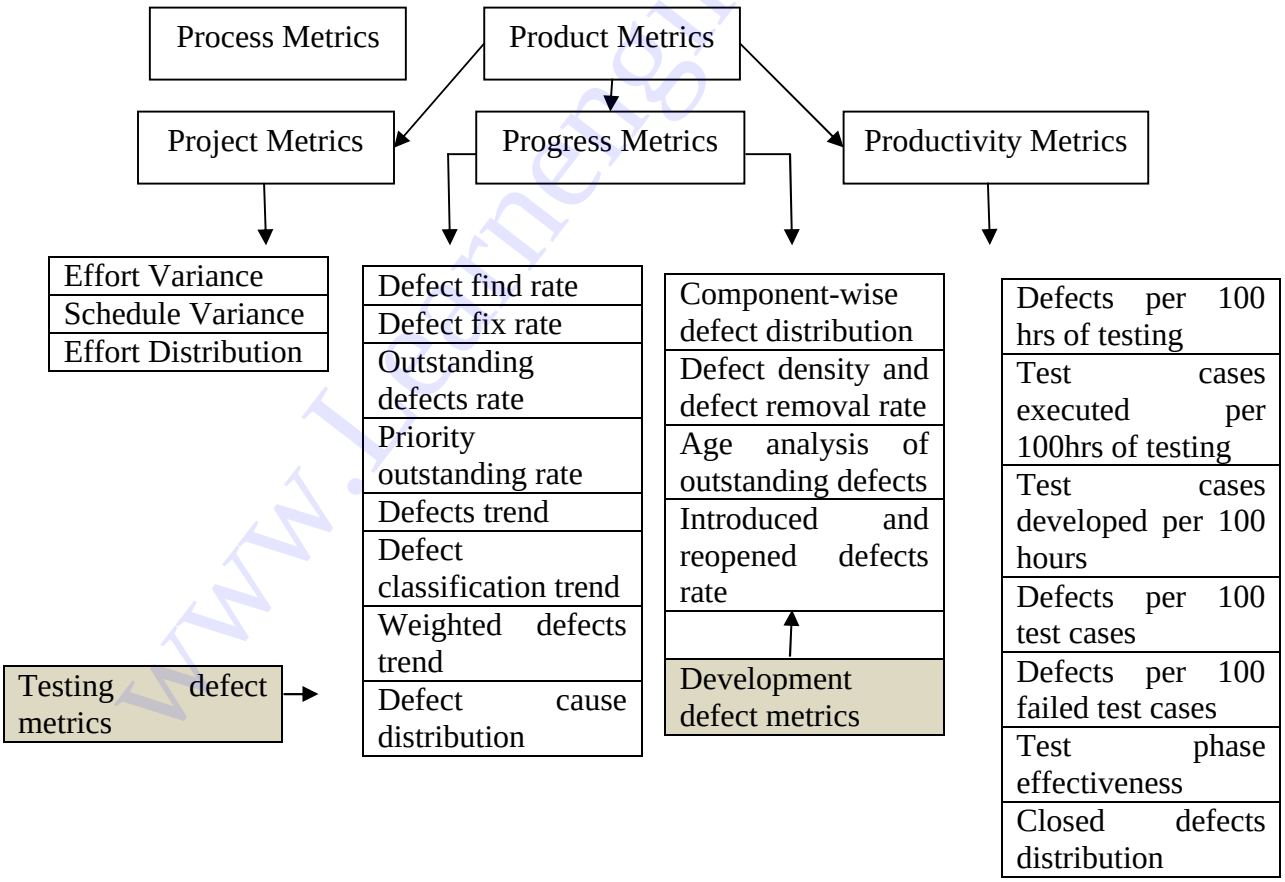
Metrics are thus derived from measurements using appropriate formulae or calculations.

Metrics in testing helps in identifying,

- When to make the release
- What to release
- Whether the product is being released with known quality

Types of Metrics

- **Project Metrics**: A set of metrics that indicates how the project is planned and executed.
- **Progress Metrics**: A set of metrics that tracks how the different activities of the project are progressing. It can be further classified into test defect metrics and development defect metrics.
- **Productivity Metrics**: A set of metrics that takes into account various productivity numbers that can be collected and used for planning and tracking testing activities.



Project Metrics

- ***Effort Variance***
  - Baselined effort estimates, revises effort estimates and actual effort are plotted together for all the phases of SDLC.
  - If there is a substantial difference between the baselined and revised effort, it points to incorrect initial estimation.
  - Calculating effort variance for each of the phases provides a quantitative measure of the relative differences between the revised and actual efforts.
  - Variance % = [(actual effort – revised estimate) / revised estimate] * 100.

- ***Schedule Variance***
  - Schedule variance is the deviation of the actual schedule from the estimated schedule. The variance percentage for schedule is same as effort.

Interpretation of ranges of effort and schedule variance

Effort variance	Schedule variance	Probable causes / results
Zero or acceptable variance	Zero variance	A well-executed project
Zero or acceptable variance	Acceptable variance	Need slight improvement in effort / schedule estimation
Unacceptable variance	Zero or acceptable variance	Underestimation; Needs further analysis
Unacceptable variance	Unacceptable variance	Underestimation of both effort and schedule
Negative variance	Zero or acceptable variance	Overestimation and schedule; both effort and schedule estimation need improvement
Negative variance	Negative variance	Overestimation and schedule; both effort and schedule estimation need improvement

- ***Effort Distribution***
  - Adequate and appropriate effort needs to be spent in each of the SDLC phase for a quality product release.
  - The distribution percentage across the different phases can be estimated at the time of planning. These can be compared with the actual at the time of release for getting a comfort feeling on the release and estimation methods.

Test Defect Metrics

Test defect metrics help us understand how the defects that are found can be used to improve testing and product quality. The defects can be classified by assigning a ***defect priority***. The priority of a defect provides a management perspective for the order of defect fixes. Also, it can

be classified as ***defect severity*** levels. The severity of defects provides the test team a perspective of the impact of that defect in product functionality.

Defect priority and defect severity – sample interpretation

Priority	What it means
1	Fix the defect on highest priority; fix it before the next build
2	Fix the defect on high priority before next test cycle
3	Fix the defect on moderate priority when time permits, before the release
4	Postpone this defect for next release or live with this defect
Severity	What it means
1	The basic product functionality failing or product crashes
2	Unexpected error condition or a functionality not working
3	A minor functionality is failing or behaves differently than expected
4	Cosmetic issue and no impact on the users
A common defect definition and classification	
Defect classification	What it means
Extreme	Product crashes or unusable Needs to be fixed immediately
Critical	Basic functionality of the product not working Needs to be fixed before next test cycle starts
Important	Extended functionality of the product not working Does not affect the progress of testing Fix it before the release
Minor	Product behave differently No impact on the test team or customers Fix it when time permits
Cosmetic	Minor irritant Need not be fixed for this release

- Defect find rate
  - To find defects early in the test cycle
- Defect fix rate
  - To fix defects as soon as they arrive
- Outstanding defects rate
  - The number of outstanding defects is very close to zero for a well-executed project all the time during the test cycle.
- Priority outstanding rate
  - Provide additional focus for those defects that mater to the release
- Defect trend
  - The effectiveness of analysis increases when several perspectives of find rate, fix rate, outstanding and priority outstanding defects are combined
- Defect classification trend
  - Providing the perspective of defect classification in the chart helps in finding out release readiness of the product.

- Weighted defect trend
  - weighted defects = [extreme*5 + critical*4 + important*3 + minor*2 + cosmetic]
  - Both large defects and large number of small defects affect product release
- Defect case distribution
  - Knowing the causes of defects helps in finding more defects and also in preventing such defects early in the cycle.

Development Defect Metrics

- Component-wise defect distribution
  - Knowing the components producing more defects helps in defect fix plan and in deciding what to release.
- Defect density and defect removal rate
  - Defects per KLOC = [Total defects found in the product] / [Total executable lines of code in KLOC]
  - Defect removal rate = [Defects found by verification activities + Defects found in unit testing] / (Defects found by test teams) * 100.
- Age analysis of outstanding defects
  - The time needed to fix a defect may be proportional to its age.
- Introduce and reopened defects trend
  - Testing is not meant to find the same defects again; release readiness should consider the quality of defect fixes.

Productivity Metrics

- Defects per 100 hours of testing
  - (Total defects found in the product for a period / Total hours spent to get those defects) * 100
  - Normalizing the defects with effort spend indicates another perspective for release quality
- Test cases executed per 100 hours of testing
  - (Total test cases executed for a period / Total hours spend in test execution) * 100
- Test cases developed per 100 hours of testing
  - (Total test cases developed for a period / Total hours spend in test case development) * 100
- Defects per 100 test cases
  - (Total defects found for a period / Total test cases executed for the same period) * 100
- Defects per 100 failed test cases
  - (Total defects found for a period / Total test cases failed due to those defects) * 100

5.7 STATUS MEETINGS, REPORTS AND CONTROL ISSUES

Measurement-related data, and other useful test-related information such as test documents and problem reports, should be collected and organized by the testing staff. The test manager can then use these items for presentation and discussion at the periodic meetings used for project monitoring and controlling. These are called ***project status meetings***.

Test-specific status meetings can also serve to monitor testing efforts, to report test progress, and to identify any test-related problems. Testers can meet separately and use test measurement data and related documents to specifically discuss test status. Following this meeting they can then participate in the overall project status meeting, or they can attend the project meetings as an integral part of the project team and present and discuss test-oriented status data at that time.

Each organization should decide how to organize and partition the meetings. Some deciding factors may be the size of the test and development teams, the nature of the project, and the scope of the testing effort.

Another type of project-monitoring meeting is the ***milestone meeting*** that occurs when a milestone has been met. A milestone meeting is a mechanism for the project team to communicate with upper management and in some cases user/client groups. Testing staff, project managers, SQA staff, and upper managers should attend.

Status meetings usually result in some type of status report published by the project manager that is distributed to upper management. Test managers should produce similar reports to inform management of test progress.

Rakos recommends that the reports be brief and contain the following items

(i) Activities and accomplishments during the reporting period: All tasks that were attended to should be listed, as well as which are complete.
(ii) Problems encountered since the last meeting period: The report should include a discussion of the types of new problems that have occurred, their probable causes, and how they impact on the project.
(iii)Problems solved: At previous reporting periods problems were reported that have now been solved. Those should be listed, as well as the solutions and the impact on the project.
(iv) Outstanding problems: These have been reported previously, but have not been solved to date. Report on any progress.
(v) Current project (testing) state versus plan: This is where graphs using process measurement data play an important role.
(vi) Expenses versus budget: Plots and graphs are used to show budgeted versus actual expenses. Earned value charts and plots are especially useful here.
(vii) Plans for the next time period: List all the activities planned for the next time period as well as the milestones.

An example bar graph for monitoring purposes can be shown with the help of a bar graph. The total number of faults found is plotted against weeks of testing effort can be shown in another graph based on defect data.

Graphs especially useful for monitoring testing costs are those that plot staff hours versus time, both actual and planned. Earned value tables and graphs are also useful. In status report graphs, earned value is usually plotted against time, and on the same graph budgeted expenses and actual expenses may also be plotted against time for comparison. Although actual expenses may be more than budget, if earned value is higher than expected, then progress may be considered satisfactory.

The agenda for a status meeting on testing includes a discussion of the work in progress since the last meeting period. Measurement data is presented, graphs are produced, and progress is evaluated. Test logs and incident reports may be examined to get a handle on the problems occurring. If there are problem areas that need attention, they are discussed and solutions are suggested to get the testing effort back on track

As testing progresses, ***status meeting attendees have to make decisions about whether to stop testing or to continue on with the testing efforts***, perhaps developing additional tests as part of the continuation process. They need to evaluate the status of the current testing efforts as compared to the expected state specified in the test plan. In order to make a decision about whether testing is complete the test team should refer to the stop test criteria included in the test plan. **If they decide that the stop-test criteria have been met, then the final status report for testing, the test summary report, should be prepared**

At ***project postmortems*** the test summary report can be used to discuss successes and failures that occurred during testing. It is a good source for test lessons learned for each project.

5.8 CRITERIA FOR TEST COMPLETION

In any event, whether progress is smooth or bumpy, at some point every project and test manager has to make the decision on when to stop testing. Since it is not possible to determine with certainty that all defects have been identified, the decision to stop testing always carries risks.

If we stop testing now, we do save resources and are able to deliver the software to our clients. However, there may be remaining defects that will cause catastrophic failures, so if we stop now we will not find them. As a consequence, clients may be unhappy with our software and may not want to do business with us in the future. Even worse there is the risk that they may take legal action against us for damages.

On the other hand, if we continue to test, perhaps there are no defects that cause failures of a high severity level. Therefore, we are wasting resources and risking our position in the market place.

Managers should not have to use guesswork to make this critical decision. The test plan should have a set of quantifiable stop-test criteria to support decision making.

The weakest stop test decision criterion is to stop testing when the project runs out of time and resources.

The stop-test criteria are as follows.

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

Fault (defect) seeding technique is based on intentionally inserting a known set of defects into a program. This provides support for a stop-test decision. It is assumed that the inserted set of defects are typical defects; that is, they are of the same type, occur at the same frequency, and have the same impact as the actual defects in the code. One way of selecting such a set of defects is to use historical defect data from past releases or similar projects.

The technique works as follow.

- Several members of the test team insert (or seed) the code under test with a known set of defects.
- The other members of the team test the code to try to reveal as many of the defects as possible.
- The number of undetected seeded defects gives an indication of the number of total defects remaining in the code (seeded plus actual).
- A ratio can be set up as follows:
    - Detected seeded defects / Detected actual defects = Total seeded defects / Total actual defects. Using this ratio we can say, for example, if the code was seeded with 100 defects and 50 have been found by the test team, it is likely that 50% of the actual defects still remain and the testing effort should continue. When all the seeded defects are found the manager has some confidence that the test efforts have been completed.

5.9 SOFTWARE CONFIGURATION MANAGEMENT

To control and monitor the testing process, testers and test mangers also need access to configuration management tools and staff. There are four major activities associated with configuration management.

1. Identification of the Configuration Items

The items that will be under configuration control must be selected, and the relationships between them must be formalized. The four configuration items are, a design specification, a test specification, an object code module, and source code module.

2. Change Control

There are two aspects of change control—one is tool-based, the other team-based. The team involved is called a configuration control board. This group oversees changes in the software system. The members of the board should be selected from SQA staff, test specialists, developers, and analysts. It is this team that oversees, gives approval for, and follows up on changes. They develop change procedures and the formats for change request forms.

3. Configuration status reporting

These reports help to monitor changes made to configuration items. They contain a history of all the changes and change information for each configuration item. Each time an approved change is made to a configuration item, a configuration status report entry is made. The reports can answer questions such as:

- who made the change;
- what was the reason for the change;
- what is the date of the change;
- what is affected by the change.

Reports for configuration items can be disturbed to project members and discussed at status meetings.

4. Configuration audits

The audit is usually conducted by the SQA group or members of the configuration control board. They focus on issues that are not covered in a technical review. A checklist of items to cover can serve as the agenda for the audit. For each configuration item the audit should cover the following:

(i) *Compliance with software engineering standards.* For example, for the source code modules, have the standards for indentation, white space, and comments been followed?
(ii) *The configuration change procedure.* Has it been followed correctly?
(iii) *Related configuration items.* Have they been updated?
(iv) *Reviews.* Has the configuration item been reviewed?

5.10 TYPES OF REVIEWS

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

The general goals for the reviewers are to:

- identify problem components or components in the software artifact that need improvement;
- identify components of the software artifact that do not need improvement;
- identify specific errors or defects in the software artifact (defect detection);
- ensure that the artifact conforms to organizational standards

To summarize, 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.

Reviews can be formal or informal. They can be technical or managerial.

Managerial reviews usually focus on project management and project status. Technical reviews are used to:

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

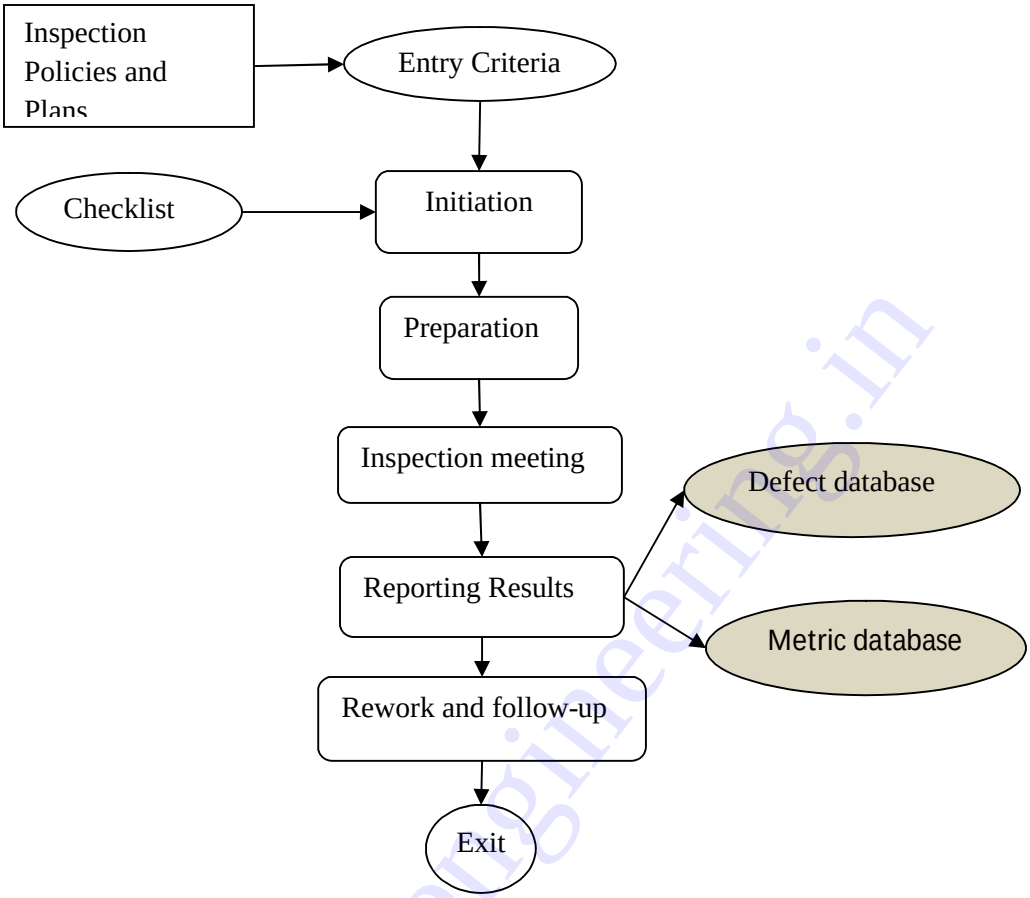
Informal reviews are an important way for colleagues to communicate and get peer input with respect to their work. There are two major types of technical reviews—inspections and walkthroughs— which are more formal in nature and occur in a meeting-like setting.

5.10.1 Inspection as a type of Technical Review

Inspections are a type of review that is formal in nature and requires pre-review preparation on the part of the review team. Several steps are involved in the inspection process as outlined in the following figure.

The ***inspection leader*** plans for the inspection, sets the date, invites the participants, distributes the required documents, runs the inspection meeting, appoints a recorder to record results, and monitors the follow up period after the review.

The checklist contains items that inspection participants should focus their attention on, check, and evaluate. The ***inspection participants*** address each item on the checklist. The ***recorde***r records any discrepancies, misunderstandings, errors, and ambiguities; in general, any problems associated with an item. The completed checklist is part of the review summary document.



When the inspection meeting has been completed (all agenda items covered) the inspectors are usually asked to sign a written document that is sometimes called a summary report.

The inspection process requires a formal follow-up process. Rework sessions should be scheduled as needed and monitored to ensure that all problems identified at the inspection meeting have been addressed and resolved. Only when all problems have been resolved and the item is either reinspected by the group or the moderator is the inspection process completed.

5.10.2 *Walkthrough as a Type of Technical Review*

Walkthroughs are a type of technical review where the producer of the reviewed material serves as the review leader and actually guides the progression of the review. If the presenter gives a skilled presentation of the material, the walkthrough participants are able to build a comprehensive mental (internal) model of the detailed design or code and are able to both evaluate its quality and detect defects.

The ***round robin review*** where there is a cycling through the review team members so that everyone gets to participate in an equal manner.

Another instance, every reviewer in a code walkthrough would lead the group in inspecting a specific line or a section of the code

5.11 DEVELOPING A REVIEW PROGRAM

The review process may be more efficient for detecting, locating, and repairing these defects, especially in the code, for the following reasons.

Reason 1

During a review there is a systematic process in place for building a real-time mental model of the software item. The reviewers step through this model building process as a group. If something unexpected appears it can be processed in the context of the real-time mental model.

There is a direct link to the incorrect, missing, superfluous item and a line/page/figure that is of current interest in the inspection. Reviewers know exactly where they are focused in the document or code and where the problem has surfaced.

They can basically carry out defect detection and defect localization tasks at the same time.

Reason 2

Reviews also have the advantage of a two-pass approach for defect detection. Pass 1 has individuals first reading the reviewed item and pass 2 has the item read by the group as a whole. If one individual reviewer did not identify a defect or a problem, others in the group are likely to find it. The group evaluation also makes false identification of defects/problems less likely. Individual testers/ developers usually work alone, and only after spending many fruitless hours trying to locate a defect will they ask a colleague for help.

Reason 3

Inspectors have the advantage of the checklist which calls their attention to specific areas that are defect prone. These are important clues. Testers/developers may not have such information available.

5.12 COMPONENTS OF A REVIEW PLAN

An organization should develop a review plan template that can be applied to all software projects. The template should specify the following items for inclusion in the review plan.

1. Review Goals

- identification of problem components or components in the software artifact that need improvement,
- identification of specific errors or defects in the software artifact,
- ensuring that the artifact conforms to organizational standards, and
- communication to the staff about the nature of the product being developed.

2. Preconditions and Items to be Reviewed

✓ items selected for review

- requirements documents;
- design documents;
- code;
- test plans (for the multiple levels);
- user manuals;
- training manuals;
- standards documents.

✓ General preconditions for a review are:

- The review of an item(s) is a required activity in the project plan.
- A statement of objectives for the review has been developed;
- The individuals responsible for developing the reviewed item indicate readiness for the review;
- The review leader believes that the item to be reviewed is sufficiently complete for the review to be useful.

3. Roles, team size, participants; time requirements;

Review Role	Responsibility
Review leader	▪ Review planning ▪ Preparing checklists ▪ Distributing review documents ▪ Managing the review meeting ▪ Issuing review reports ▪ Follow-up oversight
Review Recorder	▪ Recording and documenting problems, defects, findings, and recommendations
Reader	▪ Present review item ▪ Perform any needed rework on reviewed item
Reviewers	▪ Attend review training sessions ▪ Prepare for reviews ▪ Participate in review meetings ▪ Evaluate reviewed item ▪ Perform rework where appropriate

✓ Review Team Membership Constituency

- SQA staff Testers
- Developers
- (Analysts, designers, programmers)
- Users/clients (Optional)
- Specialists (Optional)

4. Review Procedure

For each type of review, there should be a set of standardized steps that define the given review procedure. For each step in the procedure the activities and tasks for all the reviewer participants should be defined. The review plan should refer to the standardized procedures where applicable.

5. Review Training

Review participants need training to be effective. Responsibility for reviewer training classes usually belongs to the internal technical training staff. Review participants, and especially those who will be review leaders, need the training. Test specialists should also receive review training. Suggested topics for a training program are shown below.

- ✓ Topic1. Basic concepts
- ✓ Topic 2. Review of quality issues
- ✓ Topic 3. Review of standards
- ✓ Topic 4. Understanding the material to be reviewed
- ✓ Topic 5. Defect and problem types
- ✓ Topic 6. Communication and meeting management skills
- ✓ Topic 7. Review documentation and record keeping
- ✓ Topic 8. Special instructions
- ✓ Topic 9. Practice review sessions

6. Review Checklist

A sample general review checklist for software documents

Problem/Defect Type: General Checklist

- ❖ ***Coverage and completeness***
  - ➢ **Are all essential items completed?**
  - ➢ **Have all irrelevant items been omitted?**
  - ➢ **Is the technical level of each topic addressed properly for this document?**
  - ➢ **Is there a clear statement of goals for this document? Are the goals consistent with policy?**

- ❖ ***Correctness***
  - ➢ **Are there any incorrect items?**
  - ➢ **Are there any contradictions?**
  - ➢ **Are there any ambiguities?**

- ❖ ***Clarity and Consistency***
  - ➢ **Are the material and statements in the document clear?**
  - ➢ **Are the examples clear, useful, relevant, correct?**
  - ➢ **Are the diagrams, graphs, illustrations clear, correct, use the proper notation, effective, in the proper place?**

- **Is the terminology clear, and correct?**
- **Is there a glossary of technical terms that is complete and correct?**
- **Is the writing style clear (nonambiguous)?**

❖ ***References and Aids to Document Comprehension***

- **Is there an abstract or introduction?**
- **Is there a well-placed table of contents?**
- **Are the topics or items broken down in a manner that is easy to follow and is understandable?**
- **Is there a bibliography that is clear, complete and correct?**
- **Is there an index that is clear, complete and correct?**
- **Is the page and figure numbering correct and consistent?**

5.13 REPORTING REVIEW RESULTS

Several information-rich items result from technical reviews. These items are listed below. The items can be bundled together in a single report or distributed over several distinct reports. The review reports should contain the following information.

1. *For inspections*—the group checklist with all items covered and comments relating to each item.
2. *For inspections*—a status, or summary, report (described below) signed by all participants.
3. A list of defects found, and classified by type and frequency. Each defect should be cross-referenced to the line, pages, or figure in the reviewed document where it occurs.
4. Review metric data

The inspection report on the reviewed item is a document signed by all the reviewers. It may contain a summary of defects and problems found and a list of review attendees, and some review measures. The reviewers are responsible for the quality of the information in the written report. There are several status options available,

1. *Accept:* The reviewed item is accepted in its present form or with minor rework required that does not need further verification.
2. *Conditional accept:* The reviewed item needs rework and will be accepted after the moderator has checked and verified the rework.
3. *Reinspect:* Considerable rework must be done to the reviewed item. The inspection needs to be repeated when the rework is done.

If the software item is given a conditional accept or a reinspect, a follow-up period occurs where the authors must address all the items on the problem/defect list.
The moderator reviews the rework in the case of a conditional accepts. Another inspection meeting is required to reverify the items in the case of a “reinspect” decision.

Other outputs of an inspection include a defect report and an inspection report.

The ***defect report*** contains a description of the defects, the defect type, severity level, and the location of each defect.
The ***inspection report*** contain vital data such as,

(i) number of participants in the review;
(ii) the duration of the meeting;
(iii) size of the item being reviewed (usually LOC or number of pages);
(iv) total preparation time for the inspection team;
(v) status of the reviewed item;
(vi) estimate of rework effort and the estimated date for completion of the rework.

A defect class may describe an item as missing, incorrect, or superfluous.

Defects should also be ranked in severity, for example:
(i) major (these would cause the software to fail or deviate from its specification);
(ii) minor (affects nonfunctional aspects of the software).

A ranking scale for defects can be developed in conjunction with a failure severity scale

The walkthrough report lists all the defects and deficiencies, and contains data such as:

- the walkthrough team members;
- the name of the item being examined;
- the walkthrough objectives;
- list of defects and deficiencies;
- recommendations on how to dispose of, or resolve the deficiencies

A final important item to note: The purpose of a review is to evaluate a software artifact, *not* the developer or author of the artifact. Reviews should not be used to evaluate the performance of a software analyst, developer, designer, or tester.

5.14 SOFTWARE QUALITY EVALUATION

Software quality evaluation involves defining quality attributes, quality metrics, and measurable quality goals for evaluating software work products.

Maintainability can be partitioned into the quality subfactors: testability, correctability, and expandability.

Testability is usually described as an indication of the degree of testing effort required.
Correctability is described as the degree of effort required to correct errors in the software and to handle user complaints.
Expandability is the degree of effort required to improve or modify the efficiency or functions of the software.
Functionality which is described as ‘‘an attribute that relates to the existence of certain properties and functions that satisfy stated or implied user needs.’’ It can be decomposed into the subfactors:

- **Completeness:** The degree to which the software possesses the necessary and sufficient functions to satisfy the users needs.
- **Correctness:** The degree to which the software performs its required functions.
- **Security:** The degree to which the software can detect and prevent information leak, information loss, illegal use, and system resource destruction.
- **Compatibility:** The degree to which new software can be installed without changing environments and conditions that were prepared for the replaced software.
- **Interoperability:** The degree to which the software can be connected easily with other systems and operated.

The standards document also describes a five-step methodology that guides an organization in establishing quality requirements, applying software metrics that relate to these requirements, and analyzing and validating the results.

A template called an "attribute specification format template" that can be used to clearly describe measurable system attributes. Template components are,

Software quality metrics methodology, adapted from IEEE Std 1061-1992

1. Establish software quality requirements
2. Identify the relevant software quality metrics
3. Implement the software quality metrics
4. Analyze the software quality metrics results

   Metrics that indicate low quality for software components should be subject to further scrutiny. Further scrutiny may lead to alternative conclusions, for example:
   - the software should be redesigned;
   - the software should be discarded;
   - the software should be left unchanged.
5. Validate the software quality metrics

- **Scale:** describes the scale (measurement units) that will be used for the measurement; for example, time in minutes to do a simple repair.
- **Test:** This describes the required practical tests and measurement tools needed to collect the measurement data.
- **Plan:** This is the value or level an organization plans to achieve for this quality metric. It should be of a nature that will satisfy the users/clients.
- **Best:** This is the best level that can be achieved; it may be state-of-the-art, an engineering limit for this particular development environment, but is not an expected level to be reached in this project. (An example would be a best system response time of 3.5 seconds.)
- **Worst:** This indicates the minimal level on the measurement scale for acceptance by the users/clients. Any level worse than this level indicates total system failure, no matter how good the other system attributes are. (An example would be a system response time of 6 seconds.)

- **Now:** This is the current level for this attribute in an existing system. It can be used for comparison with planned and worst levels for this project.
- **See:** This template component provides references to more detailed or related documents

Suppose a quality goal is to reach a specified level of performance. It is appropriate for testers to collect data during system test relating to:

1. **Response time.** Record time in seconds for processing and responding to a user request. (Descriptive remarks for data collection: An average value for the response time should be derived from not less than 75 representative requests, both under normal load and under stress conditions.)
2. **Memory usage.** Record number of bytes used by the application and for overhead. (Descriptive remarks for data collection: Data should be collected for normal and heavy stress and volume.)

To address quality goals such as testability, the following can be collected by the testers:

(i) **Cyclomatic complexity.**
(ii) **Number of test cases required to achieve a specified coverage goal.** Count for code structures such as statement or branch.
(iii)**Testing effort-unit test.** Record cumulative time in hours for testers to execute unit tests for an application.

For addressing goals with respect to maintainability, the following measurements are appropriate for testers to collect:

1) **Number of changes to the software.** (Descriptive remarks for data collection: Count number of problem reports)
2) **Mean time to make a change or repair.** Record mean-time-to-repair (MTTR) in time units of minutes.