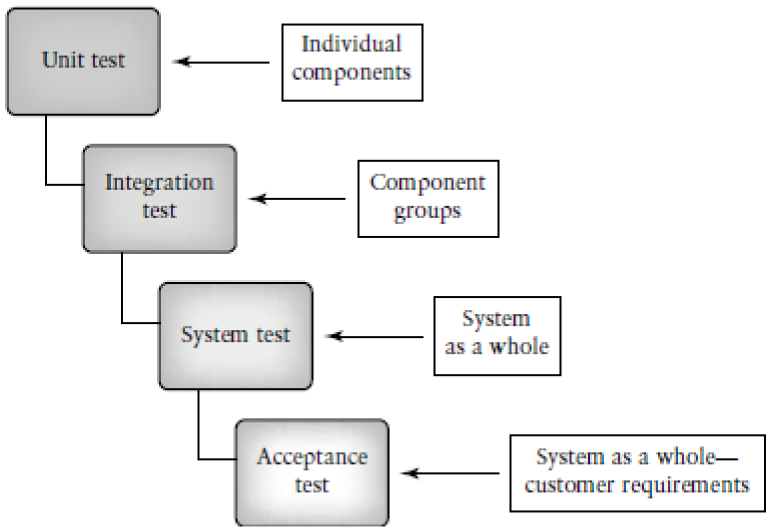
3.1 THE NEED FOR LEVELS OF TESTING

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

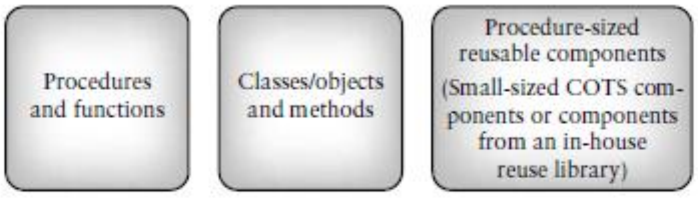
A unit is the smallest possible testable software component.

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


Procedures and functions
Classes/objects and methods
Procedure-sized reusable components (Small-sized COTS components or components from an in-house reuse library)

Some components suitable for Unit testing

Unit Testing Advantages

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

<u>Unit Test: The Need for Preparation</u>

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.

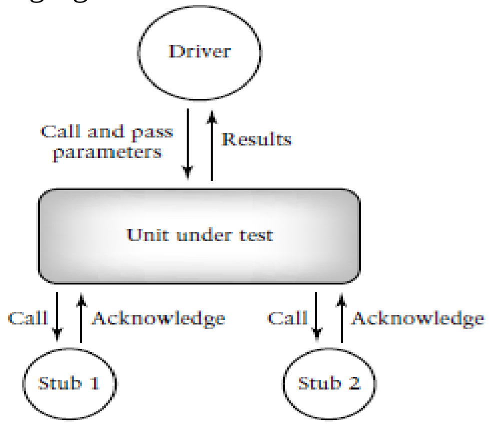
Test case design at the unit level can be based on use of the black and white box test design strategies.
Both of these approaches are useful for

- 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,

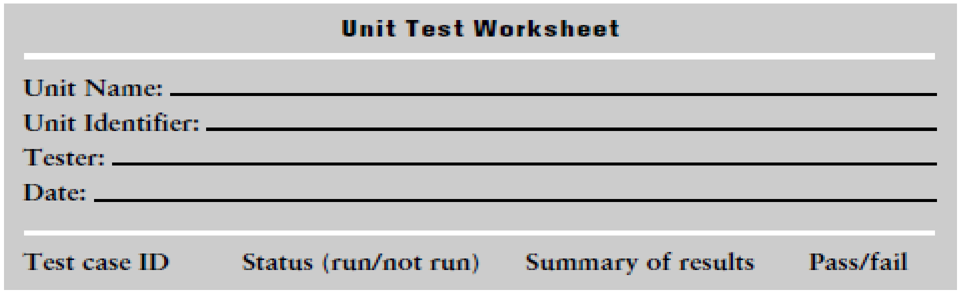
If modules D,E & F,G are the lowest module which is unit tested. Module B & C are not yet developed. The functionality of these modules is that, it calls the modules D, E & F, G. Since B and C are not yet developed, we would need some program or a "stimulator" which will call the D,E& F,G modules. These 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)

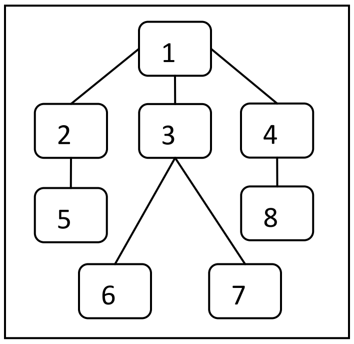
Not all interactions between the modules are known and explained through interfaces. ***Explicit interfaces*** are documented interfaces. ***Implicit interfaces*** are those which are known internally to the software engineers but are not documents
In situations where architecture or design documents 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.

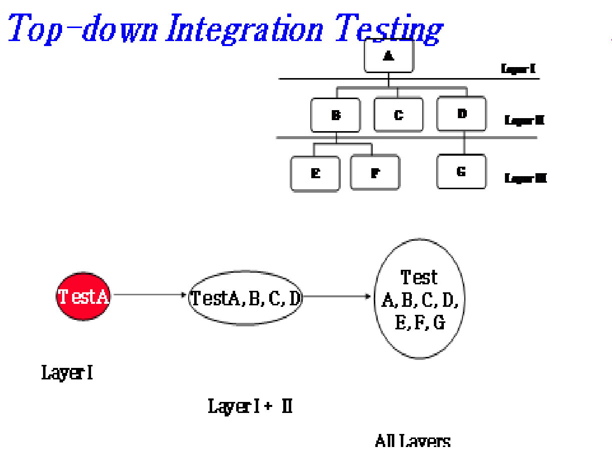
Disadvantages of Top-down Testing

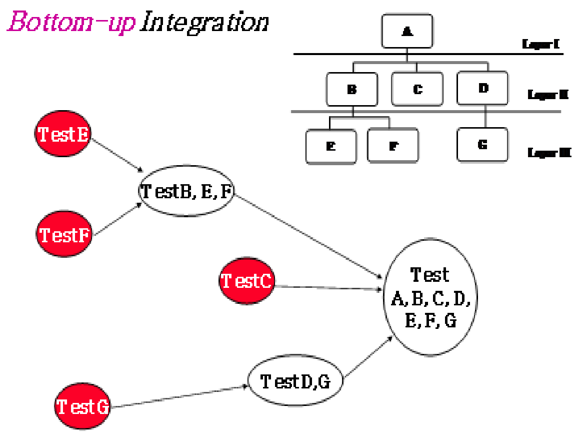
- Stubs have to be written with utmost care as they will simulate setting of output parameters.
- It is difficult to have other people or third parties to perform this testing, mostly developers will have to spend time on this.

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

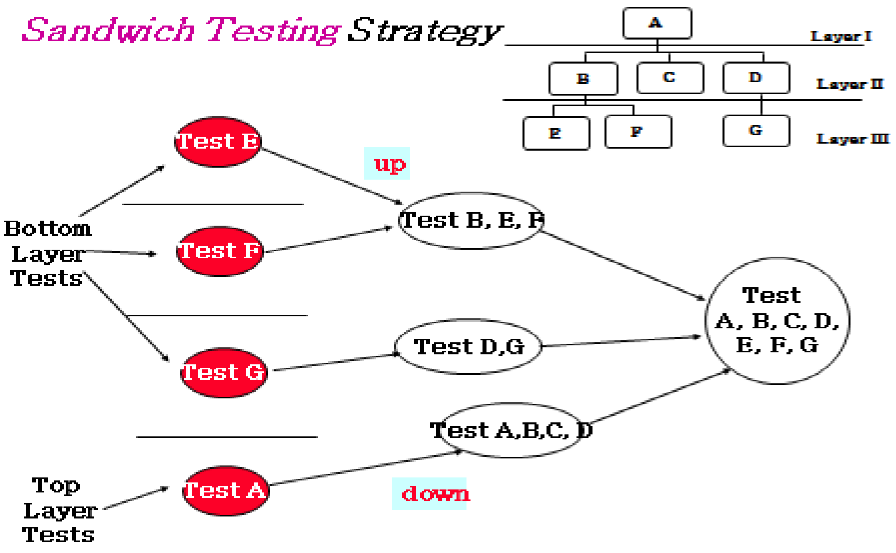
- Writing and maintaining test drivers is more difficult than writing stubs.
- This approach is not suitable for the software development using top-down approach.

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**



<u>Steps in Integration Testing (Summary)</u>

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 <u>DESIGNING INTEGRATION TESTS</u>

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,
  - o To find a large number of uncovered defects
  - o To find out system requirements (cpu, memory, disk etc)
  - o 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

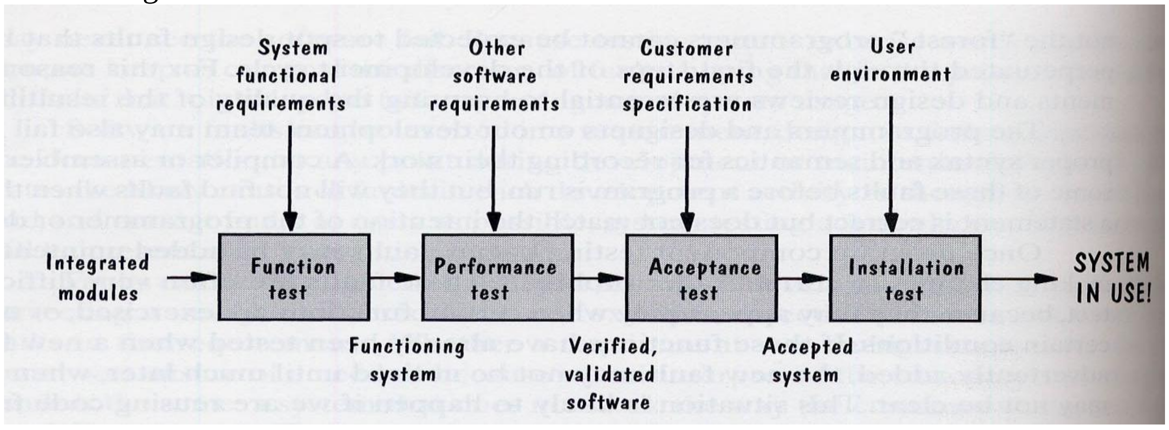
Effort saved = 600 – (A+B+C) = 600 – 480 = 120 person hours, or 20%.

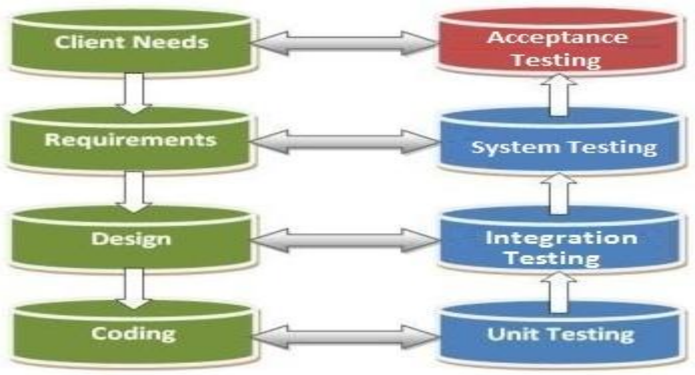
3.4 SYSTEM TESTING

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 <u>system testing.</u>

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
  - o 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.
  - o *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.]
  - o *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

<u>Best Practices in Regression Testing</u>

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

<u>Automated Regression Testing</u> 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**

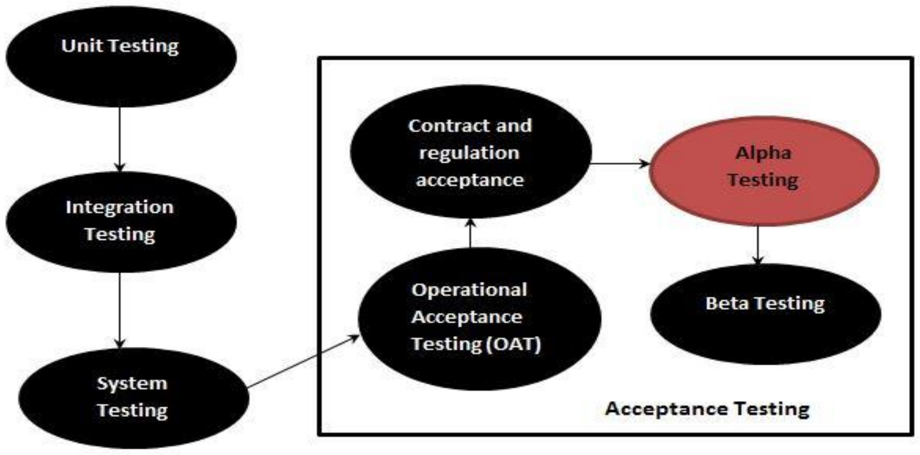
Alpha testing usually comes after system testing and involves both white and black box testing techniques. The company employees test the software for functionality and give the feedback. After this testing phase any functions and features may be added to the software.
Alpha Testing is done to ensure 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.

➢ **<u>Activities involved in the beta program</u>** 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.

- Scalability testing is performed on different configurations to check the products behavior. For example,
- If CPU utilization approaches to 100%, then another server is set up to share the load or another CPU is added to the server.
- If the results are successful, then the tests are repeated for 200 users and more to find the maximum limit for that configuration.
- 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.