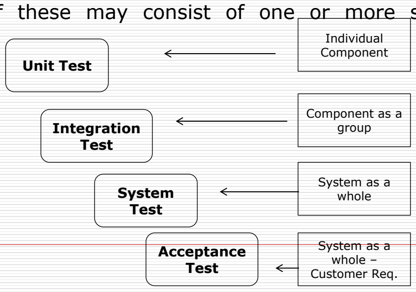
Unit II

LEVELS OF TESTING

- The Need for Levels of Testing
- Unit Test Planning –Designing the Unit Tests
- The Test Harness – Running the Unit tests and Recording Results
- Integration Tests – Designing Integration Tests
- Integration Test Planning – Scenario Testing
- Defect Bash Elimination. System Testing
- Acceptance testing – Performance testing
- Regression Testing - Internationalization testing
- Ad-hoc testing – Alpha, Beta Tests
- Testing OO systems – Usability and Accessibility Testing
- Configuration Testing - Compatibility Testing
- Testing the documentation – Website Testing
- Case Study for Unit and Integration Testing

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
- Each of these may consist of one or more sublevels or phases.



- At each level there are specific testing goals.
- For example, 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.

- An orderly progression of testing levels is described in this chapter for both object-oriented and procedural-based software systems.
- For both types of systems the testing process begins with the smallest units or components to identify functional and structural defects.
- 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. Proper interaction at the component interfaces is of special interest at the integration level.

- It usually requires the bulk of testing resources
- Laboratory equipment, special software, or special hardware may be necessary, especially for real-time, embedded, or distributed systems.
- At the system level the tester looks for defects, but the focus is on evaluating performance, usability, reliability, and other quality-related requirements.
- If the system is being custom made for an individual client then the next step following system test is acceptance test
- This is a very important testing stage for the developers.
- 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
- Software developed for the mass market (i.e., shrink-wrapped software) often goes through a series of tests called alpha and beta tests.
- Alpha tests bring potential users to the developer's site to use the software. Developers note any problems.
- Beta tests send the software out to potential users who use it under real-world conditions and report defects to the developing organization.
- Implementing all of these levels of testing require a large investment in time and organizational resources.

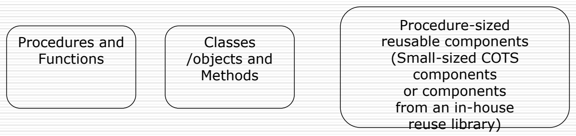
- Organizations with poor testing processes tend to skimp on resources, ignore test planning until code is close to completion, and omit one or more testing phases.
- This seldom works to the advantage of the organization or its customers

---

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

- A workable definition for a software unit is as follows:
- **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.

- A unit is traditionally viewed as a function or procedure implemented in a procedural (imperative) programming language
- A unit may also be a small-sized COTS component purchased from an outside vendor that is undergoing evaluation by the purchaser, or 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)

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

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

Unit test Planning

- 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
- brief description of a set of development phases for unit test planning is found below.

Phase 1: Describe Unit Test Approach and Risks
- In this phase of unit testing planning the general approach to unit testing is outlined. The test planner:
- (i) identifies test risks;
- (ii) describes techniques to be used for designing the test cases for the units;

- (iii) describes techniques to be used for data validation and recording of test results;
- (iv) describes the requirements for test harnesses and other software that interfaces with the units to be tested, for example, any special objects needed for testing object-oriented units.
- 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).
- Finally, 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, for 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.
- Input/output characteristics associated with each unit should also be identified, such as variables with an allowed ranges of values and performance at a certain level.

- 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
- 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.
- special tools required for the tests are also described.

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)
- (ii) the test procedures (steps required run the tests)
- Test case data should be tabularized for ease of use, and reuse
- To specifically support object-oriented test design and the organization of test data, a test case specification components of a test case is been arranged into a semantic network with parts, Object_ID, Test_Case_ID, Purpose, and List_of_Test_Case_Steps

- 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.
- All of this test design information is attached to the unit test plan
- Test cases, test procedures, and test suites may be reused from past projects if the organization has been careful to store them so that they are easily retrievable and reusable
- Both of these approaches are useful for designing test cases for functions and procedures.
- They are also useful for designing tests for the individual methods (member functions) contained in a class

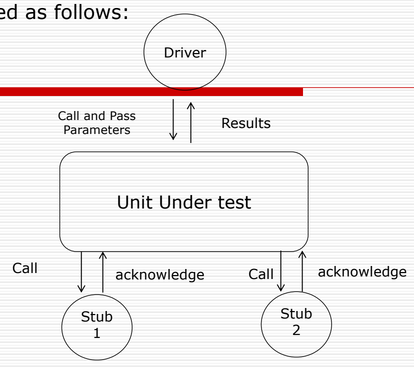
- Considering the relatively small size of a unit, it makes sense to focus on white box test design for procedures/ functions and the methods in a class.
- This approach gives the tester the opportunity to exercise logic structures and/or data flow sequences, with the goal of evaluating the structural integrity of the unit.
- Some black box–based testing is also done at unit level; however, the bulk of black box testing is usually done at the integration and system levels and beyond
- In the case of a smaller-sized COTS component selected for unit testing, a black box test design approach may be the only option.

- It should be mentioned that for units that perform mission/safely/business critical functions, it is often useful and prudent to design stress, security, and performance tests at the unit level if possible

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
- This code called the test harness, is developed especially for test and is in addition to the code that composes the system under development
- **"The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls."**

- The role is of the test harness is shown in Figure and it is defined as follows:



- The development of drivers and stubs requires testing resources
- The drivers and stubs must be tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software.
- Drivers and stubs can be developed at several levels of functionality
- For example,
- a driver could have the following options and combinations of options:
  - **(i)** call the target unit;
  - **(ii)** do 1, and pass inputs parameters from a table;
  - **(iii)** do 1, 2, and display parameters;
  - **(iv)** do 1, 2, 3 and display results (output parameters)

- The stubs could also exhibit different levels of functionality. For example a stub could:
- (i) display a message that it has been called by the target unit;
- (ii) do 1, and display any input parameters passed from the target unit;
- (iii) do 1, 2, and pass back a result from a table;
- (iv) do 1, 2, 3, and display result from table

Stub	Driver
Created by the Team of testers during the process of Top-down testing	Drivers are used when the main module is not ready and complex than stubs and developed during bottom up approach
Stubs are commonly referred to as called programs	Drivers are calling programs
The purpose of the stub is to test the upper levels of the code when lower levels are not yet developed	The purpose of the code is to test the lower levels of the code when upper levels of the coder are not yet developed
Stubs and Drivers both work as a substitute for a missing or unavailable module	

Running the Unit tests and Recording the Results

- Unit tests can begin when
  - (i) the units becomes available from the developers (an estimation of availability is part of the test plan)
  - (ii) the test cases have been designed and reviewed, and
  - (iii) 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 Table

Unit test Worksheet			
Unit Name :			
Unit Identifier :			
Tester :			
Date :			
Test Case ID	Status (run/ not run)	Summary of Results	Pass / Fail

- These forms can be included in the test summary report, and are of value at the weekly status meetings that are often used to monitor test progress.
- 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
- Differences from expected behavior should be described in detail.
- This gives clues to the developers to help them locate any faults.
- During testing the tester may determine that additional tests are required

- For example, a tester may observe that a particular coverage goal has not been achieved. The test set will have to be augmented and the test plan documents should reflect these changes.
- 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).
- Other likely causes that need to be carefully investigated by the tester are the following

- 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.
- When testing of the units is complete, a test summary report should be prepared.
- This is a valuable document for the groups responsible for integration and system tests

Integration Test: Goals

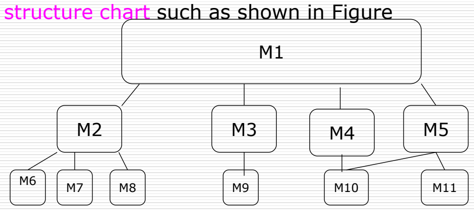
- Integration test for procedural code has two major goals:
- (i) to detect defects that occur on the interfaces of units;
- (ii) to assemble the individual units into working subsystems and finally a complete system that is ready for system test.
- In unit test the testers attempt to detect defects that are related to the functionality and structure of the unit.
- However, the interfaces are more adequately tested during integration test when each unit is finally connected to a full and working implementation of those units it calls, and those that call it.

- As a consequence of this assembly or integration process, software subsystems and finally a completed system is put together during the integration test
- completed system is then ready for system testing.
- Integration testing works best as an iterative process in procedural oriented system.
- One unit at a time is integrated into a set of previously integrated modules which have passed a set of integration tests.
- The interfaces and functionally of the new unit in combination with the previously integrated units is tested.
- When a subsystem is built from units integrated in this stepwise manner, then performance, security, and stress tests can be performed on this subsystem.

- The integration process in object-oriented systems is driven by assembly of the classes into cooperating groups
- The cooperating groups of classes are tested as a whole and then combined into higher-level groups.
- Usually the simpler groups are tested first, and then combined to form higher-level groups until the system is assembled.

Integration Strategies for Procedures and Functions

- For conventional procedural/functional-oriented systems there are two major integration strategies—top-down and bottom-up.
- In both of these strategies only one module at a time is added to the growing subsystem.
- To plan the order of integration of the modules in such system a structure chart such as shown in Figure



- Structure charts, or call graphs as they are otherwise known, are used to guide integration
- These charts show hierarchical calling relationships between modules.
- Each node, or rectangle in a structure chart, represents a module or unit, and the edges or lines between them represent calls between the units.
- In the simple chart the rectangles M1– M11 represent all the system modules.
- Again, a line or edge from an upper-level module to one below it indicates that the upper level module calls the lower module.

- Bottom-up integration of the modules begins with testing the lowest level modules, those at the bottom of the structure chart
- These are modules that do not call other modules.
- In the structure chart example these are modules M6, M7, M8, M9, M10, M11.
- Drivers are needed to test these modules.
- The next step is to integrate modules on the next upper level of the structure chart whose subordinate modules have already been tested
- For example, if we have tested M6, M7, and M8, then we can select M2 and integrate it with M6, M7, and M8. The actual M2 replaces the drivers for these modules.

- Accordingly we can integrate M9 with M3 when M9 is tested, and M4 with M10 when M10 is tested, and finally M5 with M11 and M10 when they are both tested.
- Integration of the M2 and M3 subsystems can be done in parallel by two testers
- The M4 and M5 subsystems have overlapping dependencies on M10.
- To complete the subsystem represented by M5, both M10 and M11 will have to be tested and integrated. M4 is only dependent on M10. A third tester could work on the M4 and M5 subsystems

- After this level of integration is completed, we can then move up a level and integrate the subsystem M2, M6, M7, and M8 with M1
- The same conditions hold for integrating the subsystems represented by M3, M9, M4, M10, and M5, M10, M11 with M1.
- In that way the system is finally integrated as a whole. In this example a particular sequence of integration has been selected.
- A rule of thumb for bottom-up integration says that to be eligible for selection as the next candidate for integration, all of a module's subordinate modules (modules it calls) must have been tested previously.

- Top-down integration starts at the top of the module hierarchy
- The rule of thumb for selecting candidates for the integration sequence says that when choosing a candidate module to be integrated next, at least one of the module's superordinate (calling) modules must have been previously tested.
- In our case, M1 is the highest-level module
- We create four stubs to represent M2, M3, M4, and M5. When the tests are passed, then we replace the four stubs by the actual modules one at a time.
- When we have integrated the modules M2–M5, then we can integrate the lowest-level modules.

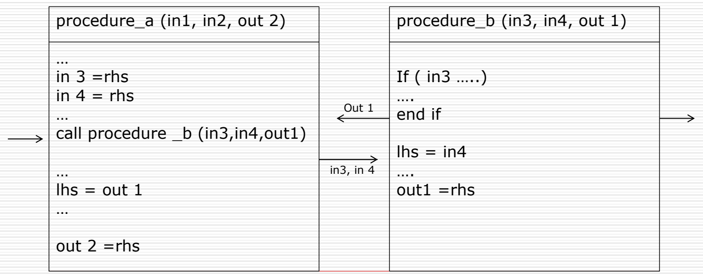
- One can traverse the structure chart and integrate the modules in a depth or breadth-first manner.
- For example, the order of integration for a depth-first approach would be M1, M2, M6, M7, M8, M3, M9, M4, M10, M5, M11.
- Breadth-first would proceed as M1, M2, M3, M4, M5, M6, M7, M8, M9, M10, M11.
- Note that using the depth-first approach gradually forms subsystems as the integration progresses
- Top-down integration ensures that the upper-level modules are tested early in integration. If they are complex and need to be redesigned there will be more time to do so.

- **Integration Strategies for Classes**
- For object-oriented systems the concept of a structure chart and hierarchical calling relationships are not applicable
- Therefore, integration needs to proceed in a manner different from described previously
- A good approach to integration of an object-oriented system is to make use of the concept of object clusters
- Clusters are somewhat analogous to small subsystems in procedural-oriented systems.

Designing Integration Tests

- Integration tests for procedural software can be designed using a black or white box approach
- Since many errors occur at module interfaces, test designers need to focus on exercising all input/output parameter pairs, and all calling relationships.
- 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.

- Procedure_b is being integrated with Procedure_a. Procedure_a calls Procedure_b with two input parameters in3, in4.
- Procedure_b uses those parameters and then returns a value for the output parameter out1.



- Terms such as lhs and rhs could be any variable or expression.
- The reader should interpret the use of the variables in the broadest sense. The parameters could be involved in a number of def and/or use data flow patterns
- Data flow–based (def-use paths) and control flow (branch coverage) test data generation methods are useful here to insure that the input parameters, in3, in4, are used properly in Procedure_b.
- Again data flow methods (def-use pairs) could also be used to check that the proper sequence of data flow operations is being carried out to generate the correct value for out1 that flows back to procedure_a.

- Black box tests are useful in this example for checking the behavior of the pair of procedures
- For this example test input values for the input parameters in1 and in2 should be provided, and the outcome in out2 should be examined for correctness
- 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.
- The reader can visualize these as coverage criteria for integration test.
- Some black box tests used for module integration may be reusable from unit testing
- 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

- 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.
- The tester also wants to observe subsystem actions when excessive amounts of transactions are generated
- Performance issues such as the time requirements for a transaction should also be subjected to test

- 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
- In object-oriented testing framework the method is the entity selected for unit test
- A group of cooperating classes is selected for test as a cluster
- In their object-oriented testing framework the method is the entity selected for unit test
- The methods and the classes they belong to are connected into clusters of classes that are represented by a directed graph that has two special types of entities.
- These are method-message paths, and atomic systems functions that represent input port events

- 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
- Some examples of clusters are groups of classes that produce a report, or monitor and control a device

Integration Test Planning

- 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, all necessary to plan integration tests

- For procedural-oriented system the order of integration of the units 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
- In addition, testing resources and schedules for integration should be included in the test plan
- For readers integrating object-oriented systems has a detailed description of a Cluster Test Plan

(i) clusters this cluster is dependent on;
(ii) a natural language description of the functionality of the cluster to be tested;
(iii) list of classes in the cluster;
(iv) a set of cluster test cases.

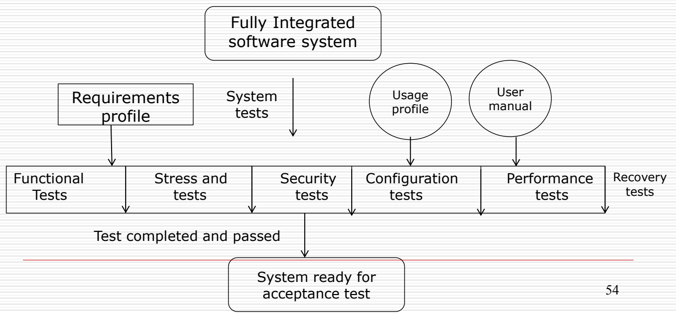
- one of the goals of integration test is to build working subsystems, and then combine these into the system as a whole
- When planning for integration test the planner selects subsystems to build based upon the requirements and user needs
- Very often subsystems selected for integration are prioritized.

- System Test: The Different Types
- When integration tests are completed, a software system has been assembled and its major subsystems have been tested
- At this point the developers/ testers begin to test it as a whole
- System test planning should begin at the requirements phase with the development of a master test plan and requirements-based (black box) tests.
- System test planning is a complicated task. There are many components of the plan that need to be prepared such as test approaches, costs, schedules, test cases, and test procedures.

- System testing itself requires a large amount of resources. The goal is to ensure that the system performs according to its requirements.
- System test evaluates both functional behavior and quality requirements such as reliability, usability, performance and security.
- This phase of testing is especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage.

- After system test the software will be turned over to users for evaluation during acceptance test or alpha/beta test.
- The organization will want to be sure that the quality of the software has been measured and evaluated before users/clients are invited to use the system
- The team must do their best to find any weak areas in the software; therefore, it is best that no developers are directly involved.
- There are several types of system tests as shown on Figure The types are as follows:
- Functional testing
- Performance testing
- Stress testing
- Configuration testing
- Security testing
- Recovery testing

- Not all software systems need to undergo all the types of system testing.
- Test planners need to decide on the type of tests applicable to a particular software system.
- Decisions depend on the characteristics of the system and the available test resources



- For example, if multiple device configurations are not a requirement for your system, then the need for configuration test is not significant

- Test resources can be used for other types of system tests. Figure also shows some of the documents useful for system test design, such as the requirements document, usage profile, and user manuals. For both procedural- and object-oriented systems, use cases, if available, are also helpful for system test design
- During system test the testers can repeat these tests and design additional tests for the system as a whole

- The repeated tests can in some cases be considered regression tests since there most probably have been changes made to the requirements and to the system itself since the initiation of the project
- An important tool for implementing system tests is a load generator.
- A load generator is essential for testing quality requirements such as performance and stress
- **A load is a series of inputs that simulates a group of transactions.**
- **A transaction is a unit of work seen from the system user's view**

- A transaction consists of a set of operations that may be performed by a person, software system, or a device that is outside the system
- If you were system testing a telecommunication system you would need a load that simulated a series of phone calls (transactions) of particular types and lengths arriving from different locations
- A load can be a real load, that is, you could put the system under test to real usage by having actual telephone users connected to it.
- Loads can also be produced by tools called load generators
- They will generate test input data for system test. Load generators can be simple tools that output a fixed set of predetermined transactions

- They can be complex tools that use statistical patterns to generate input data or simulate complex environments. Users of the load generators can usually set various parameters.
- For example, in our telecommunication system load generator users can set parameters for the mean arrival rate of the calls, the call durations, the number of wrong numbers and misdials, and the call destinations.

Functional Testing

- System functional tests have a great deal of overlap with acceptance tests.
- Very often the same test sets can apply to both
- Both are demonstrations of the system's functionality
- Functional tests at the system level are used to ensure that the behavior of the system adheres to the requirements specification.
- All functional requirements for the system must be achievable by the system.

- For example, if a personal finance system is required to allow users to set up accounts, add, modify, and delete entries in the accounts, and print reports, the function-based system and acceptance tests must ensure that the system can perform these tasks
- Functional tests are black box in nature
- The focus is on the inputs and proper outputs for each function.
- Improper and illegal inputs must also be handled by the system.
- System behaviour under the latter circumstances tests must be observed. All functions must be tested.

- Many of the system-level tests including functional tests should be designed at requirements time, and be included in the master and system test plans
- Since functional tests are black box in nature, equivalence class partitioning and boundary-value analysis are useful methods that can be used to generate test cases. State-based tests are also valuable.
- In fact, the tests should focus on the following goals
  - All types or classes of legal inputs must be accepted by the software.
  - All classes of illegal inputs must be rejected (however, the system should remain available).
  - All possible classes of system output must exercised and examined.
  - All effective system states and state transitions must be exercised and examined.
  - All functions must be exercised

- Defined and documented form should be used for recording test results from functional and all other system tests.
- If a failure is observed, a formal test incident report should be completed and returned with the test log to the developer for code repair
- Managers keep track of these forms and reports for quality assurance purposes, and to track the progress of the testing process.

Performance Testing

- An examination of a requirements document shows that there are two major types of requirements
  - Functional requirements. Users describe what functions the software should perform. We test for compliance of these requirements at the system level with the functional-based system tests.
  - Quality requirements. There are non functional in nature but describe quality levels expected for the software. One example of a quality requirement is performance level. The users may have objectives for the software system in terms of memory use, response time, throughput, and delays.

- The goal of system performance tests is to see if the software meets the performance requirements. Testers also learn from performance test
- whether there are any hardware or software factors that impact on the system's performance
- Performance testing allows testers to tune the system; that is, to optimize the allocation of system resources.
- For example, testers may find that they need to reallocate memory pools, or to modify the priority level of certain system operations.
- Testers may also be able to project the system's future performance levels. This is useful for planning subsequent releases.

- Performance objectives must be articulated clearly by the users /clients in the requirements documents, and be stated clearly in the system test plan.
- The objectives must be quantified
- For example, a requirement that the system return a response to a query in "a reasonable amount of time" is not an acceptable requirement; the time requirement must be specified in quantitative way.
- Results of performance tests are quantifiable
- At the end of the tests the tester will know, for example, the number of CPU cycles used, the actual response time in seconds (minutes, etc.), the actual number of transactions processed per time period

- Resources for performance testing must be allocated in the system test plan.
- Among the resources are:
- A source of transactions to drive the experiments. For example if you were performance testing an operating system you need a stream of data that represents typical user interactions. Typically the source of transaction for many systems is a load generator (as described in the previous section).

- An experimental testbed that includes hardware and software the system-under-test interacts with. The testbed requirements sometimes include special laboratory equipment and space that must be reserved for the tests.
- Instrumentation or probes that help to collect the performance data. Probes may be hardware or software in nature. Some probe tasks are event counting and event duration measurement. For example, if you are investigating memory requirements for your software you could use a hardware probe that collected information on memory usage (blocks allocated, blocks deallocated for different types of memory per unit time) as the system executes.

The tester must keep in mind that the probes themselves may have an impact on system performance.

- A set of tools to collect, store, process, and interpret the data. Very often, large volumes of data are collected, and without tools the testers may have difficulty in processing and analyzing the data in order to evaluate true performance levels.

Test managers should ascertain the availability of these resources, and allocate the necessary time for training in the test plan. Usage requirements for these resources need to be described as part of the test plan

- **S t r e s s T e s t i n g**
- When a system is tested with a load that causes it to allocate its resources in maximum amounts, this is called stress testing
- For example, if an operating system is required to handle 10 interrupts/second and the load causes 20 interrupts/second, the system is being stressed. The goal of stress test is to try to break the system; find the circumstances under which it will crash. This is sometimes called "breaking the system.

- Stress testing is important because it can reveal defects in real-time and other types of systems, as well as weak areas where poor design could cause unavailability of service
- For example, system prioritization orders may not be correct, transaction processing may be poorly designed and waste memory space, and timing sequences may not be appropriate for the required tasks
- This is particularly important for real-time systems where unpredictable events may occur resulting in input loads that exceed those described in the requirements documents

- Stress testing often uncovers race conditions, deadlocks, depletion of resources in unusual or unplanned patterns, and upsets in normal operation of the software system.
- All of these conditions are likely to reveal defects and design flaws which may not be revealed under normal testing conditions.
- The reader should note that stress tests should also be conducted at the integration, and if applicable at the unit level, to detect stress-related defects as early as possible in the testing process

- C o n f i g u r a t i o n T e s t i n g
- Typical software systems interact with hardware devices such as disc drives, tape drives, and printers
- Many software systems also interact with multiple CPUs, some of which are redundant.
- Software that controls real time processes, or embedded software also interfaces with devices, but these are very specialized hardware items such as missile launchers, and nuclear power device sensors.

- In many cases, users require that devices be interchangeable, removable, or reconfigurable
- For example, a printer of type X should be substitutable for a printer of type Y, CPU A should be removable from a system composed of several other CPUs, sensor A should be replaceable with sensor B.
- Very often the software will have a set of commands, or menus, that allows users to make these configuration changes.
- Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur

---

- According to Beizer configuration testing has the following objectives
  - Show that all the configuration changing commands and menus work properly.
  - Show that all interchangeable devices are really interchangeable, and that they each enter the proper states for the specified conditions.
  - Show that the systems' performance level is maintained when devices are interchanged, or when they fail.
- Several types of operations should be performed during configuration test. Some sample operations for testers are

---

  - (i) rotate and permutate the positions of devices to ensure physical/ logical device permutations work for each device (e.g., if there are two printers A and B, exchange their positions);
  - (ii) induce malfunctions in each device, to see if the system properly handles the malfunction;
  - (iii) induce multiple device malfunctions to see how the system reacts.
- These operations will help to reveal problems (defects) relating to hardware/ software interactions when hardware exchanges, and reconfigurations occur. Testers observe the consequences of these operations and determine whether the system can recover gracefully particularly in the case of a malfunction.

- S e c u r i t y T e s t i n g
- Designing and testing software systems to insure that they are safe and secure is a big issue facing software developers and test specialists
- Recently, safety and security issues have taken on additional importance due to the proliferation of commercial applications for use on the Internet.
- If Internet users believe that their personal information is not secure and is available to those with intent to do harm, the future of e-commerce is in peril!

- Security testing evaluates system characteristics that relate to the availability, integrity, and confidentially of system data and services.
- Computer software and data can be compromised by:
  - criminals intent on doing damage, stealing data and information, causing denial of service, invading privacy;
  - errors on the part of honest developers/maintainers who modify, destroy, or compromise data because of misinformation, misunderstandings, and/or lack of knowledge

---

- Both criminal behavior and errors that do damage can be perpetuated by those inside and outside of an organization
- Damage can be done through various means such as
  - (i) viruses;
  - (ii) trojan horses;
  - (iii) trap doors;
  - (iv) illicit channels
- The effects of security breaches could be extensive and can cause:

  - (i) loss of information;
  - (ii) corruption of information;
  - (iii) misinformation;
  - (iv) privacy violations;
  - (v) denial of service.
- Developers try to ensure the security of their systems through use of protection mechanisms such as passwords, encryption, virus checkers, and the detection and elimination of trap doors.
- Password checking and examples of other areas to focus on during security testing are described below.

- Password Checking—Test the password checker to insure that users will select a password that meets the conditions described in the password checker specification. Equivalence class partitioning and boundary value analysis based on the rules and conditions that specify a valid password can be used to design the tests.
- Legal and Illegal Entry with Passwords—Test for legal and illegal system/data access via legal and illegal passwords

---

- Password Expiration—If it is decided that passwords will expire after a certain time period, tests should be designed to insure the expiration period is properly supported and that users can enter a new and appropriate password.
- Encryption—Design test cases to evaluate the correctness of both encryption and decryption algorithms for systems where data/messages are encoded
- Browsing—Evaluate browsing privileges to insure that unauthorized browsing does not occur. Testers should attempt to browse illegally and observe system responses. They should determine what types of private information can be inferred by both legal and illegal browsing.

- Trap Doors—Identify any unprotected entries into the system that may allow access through unexpected channels (trap doors). Design tests that attempt to gain illegal entry and observe results. Testers will need the support of designers and developers for this task. In many cases an external "tiger team" as described below is hired to attempt such a break into the system.
- Viruses—Design tests to insure that system virus checkers prevent or curtail entry of viruses into the system. Testers may attempt to infect the system with various viruses and observe the system response. If a virus does penetrate the system, testers will want to determine what has been damaged and to what extent.

- security is an especially important issue, as in the case of network software, then the best approach if resources permit, is to hire a so-called "tiger team" which is an outside group of penetration experts who attempt to breach the system security
- Although a testing group in the organization can be involved in testing for security breaches, the tiger team can attack the problem from a different point of view.
- Before the tiger team starts its work the system should be thoroughly tested at all levels.
- The testing team should also try to identify any trap doors and other vulnerable points. Even with the use of a tiger team there is never any guarantee that the software is totally secure.

- R e c o v e r y T e s t i n g
- Recovery testing subjects a system to losses of resources in order to determine if it can recover properly from these losses
- This type of testing is especially important for transaction systems, for example, on-line banking software.
- Tests would determine if the system could return to a well known state, and that no transactions have been compromised.

- They usually have multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device.
- They also have a so-called "checkpoint" system that meticulously records transactions and system states periodically so that these are preserved in case of failure
- The recovery testers must ensure that the device monitoring system and the checkpoint software are working properly.

- Beizer advises that testers focus on the following areas during recovery testing
  - 1. Restart. The current system state and transaction states are discarded. The most recent checkpoint record is retrieved and the system initialized to the states in the checkpoint record. Testers must insure that all transactions have been reconstructed correctly and that all devices are in the proper state. The system should then be able to begin to process new transactions.
  - 2. Switchover. The ability of the system to switch to a new processor must be tested. Switchover is the result of a command or a detection of a faulty processor by a monitor

- In each of these testing situations all transactions and processes must be carefully examined to detect:
  - i) loss of transactions;
  - (ii) merging of transactions;
  - (iii) incorrect transactions;
  - (iv) an unnecessary duplication of a transaction.
- A good way to expose such problems is to perform recovery testing under a stressful load. Transaction inaccuracies and system crashes are likely to occur with the result that defects and design flaws will be revealed.

- Regression Testing
- Regression testing is not a level of testing, but it is the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes
- Regression testing can occur at any level of test,
- The unit is repaired and then retested with all the old test cases to ensure that the changes have not affected its functionality

- Regression tests are especially important when multiple software releases are developed
- Users want new capabilities in the latest releases, but still expect the older capabilities to remain in place.
- This is where regression testing plays a role.
- Automated testing tools support testers with this very time-consuming task.

- It is important to ensure that
  - The changes or additions work as designed
  - The changes or additions do not break something that is already working and should continue to work
- Regression testing follows selective re-testing technique.
  - Whenever defect fixes are done, a set of test cases need to be run to verify the defect fixes
  - An impact analysis is done to find what areas may get impacted due to defect fixes
  - Based on the impact analysis, some more test cases are selected to take care of the impacted areas
  - Since this technique focuses on reuse of existing test cases that , this technique is called selective re-testing.

Types of Regression testing

- A build is an aggregation of all the defect fixes and features that are present in the product.
- There are two types of regression testing
  - Regular regression testing
  - Final regression testing
- A regular regression testing is done between test cycles to ensure that the defect fixes are done and the functionality that were working continue to work
- A final regression testing is done to validate the final build before release. It is conducted for a specific period of duration.

When to do Regression Testing ?

- It is necessary to perform regression testing when
  - A reasonable amount of initial testing is already carried out
  - A good number of defects have been fixed
  - Defect fixes that can produce side-effects are taken care of
- Regression testing also be performed periodically as a pro-active measure.

How to do Regression testing?

- The objective of this is to explain methodology that encompasses the majority of them. The methodology here is made of the following steps
  - Performing an initial smoke or sanity test
  - Understanding the criteria for selecting the test cases
  - Classifying the test cases into different priorities
  - A methodology for selecting test cases
  - Resetting the test cases for test execution
  - Concluding the results of a regression cycle

Performing smoke or sanity test

- Smoke test consists of
  - 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 and
  - If this suite fails, developers to identify the changes and rollback the changes to a state where the smoke test suite succeeds.

Understanding the criteria for selecting test cases

- First, an organization choose to have a constant set of regression tests run for every build or change. But this approach is likely to suboptimal bcoz
  - In order to cover all fixes, the constant set of test will encompass all features and tests which are not required may be run every time.
  - A given set of defect fixes may introduce problems for which there may not be ready-made test cases in the constant set.
- A second approach is to select the test cases dynamically for each build by making judicious choices of the test cases.

Classifying test cases

- To enable choosing right tests for a regression run, the test cases can be classified into various priorities based on importance and customer usage
- Priority 0 : These test cases can be called sanity test cases which check basic functionality and are run for accepting the build for further testing
- Priority 1 : Uses the basic and normal setup and these test cases deliver high project value to both development team and to users.
- Priority 2 : These test cases deliver moderate project value. They are executed as part of the testing cycle and selected for regression testing on a need basis.

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 (TCDB)
- Case 2 : If the criticality and the impact of the defect fixes are medium, then we need to execute all priority -0 and priority -1 test cases.
- Case 3 : If the criticality and impact of the defect fixes are high, then we need to execute all Priority -0, priority-1, and carefully selected subset of Priority -2 test cases.
- Alternative methodologies are
  - Regress all
  - Priority based regression
  - Regress changes – code changes are compared to the last cycle
  - Random regression – random test cases are selected and executed

Resetting the test cases for regression testing

- Resetting test cases reduces the risk involved in testing defect fixes by making the testers go through all the test cases and selecting appropriate test cases based on the impact of those defect fixes.

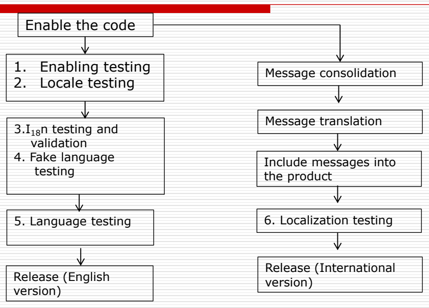
Concluding the results of regression testing

Current result from regression	Previous results	Conclusion	Remarks
Fail	Pass	Fail	Need to improve the regression process and code reviews
Pass	Fail	Pass	Good regression-defect fixes work
Fail	Fail	Fail	Need to analyze why defect fixes are not working
Pass (with a work around)	Fail	Analyze the work around and if satisfied mark result as PASS	Work around needs a good review as they can create side effects
Pass	Pass	Pass	Good result

Internationalization Testing

- Building software for the International market, supporting multiple languages, in a cost-effective and timely manner is a matter of using internationalization standards throughout the software development life cycle – from requirements capture though design, development, testing and maintenance.
- Character set
  - The purpose is to introduce the idea of different character representations
  - ASCII – American Standard Code for Information Interchange. It is a byte representation – 128 characters (2^7), Later introduced 256 characters (2^8) – enabled special characters, punctuation, symbols etc ASCII also helped accented characters to be represented (Ex : naeiou) - english like characters
- Double-byte character set (DBCS) – Some languages such as Chinese and Japanese have many different characters that cannot be represented in a byte. They use two bytes to represent each character.
- Unicode : ASCII and DBCS not sufficient to represent all the characters of all the languages in the world. The characters for all the languages need to be stored, interpreted, and transmitted in a standard way. Unicode fill this need effectively.

Test Phases for Internationalization Testing



- Enabling Testing
  - An activity of code review or code inspection for unit testing, with an objective to catch $I_{18}n$ defects is called enabling testing
    - Check the code for APIs function calls that are not part of the I18n API set.
    - Check the code for hard-coded date, currency formats
    - Check the code to see that there are no computations
    - Check the dialog boxes to see whether they leave atleast 0.5 times more space for expansion
    - Ensure region-cultural based messages and slang are not in the code
- Locale Testing
  - Changing the different locales using the system settings or environment variables and testing the software functionality, number, date, time and currency format is locale testing

Internationalization validation

- Objectives are
  - Software is tested for functionality with ASCII, DBCS and European characters
  - The software handles string operations,
  - The software display is consistent with characters which are non-ASCII in GUI and menus.
  - The software messages are handled properly.

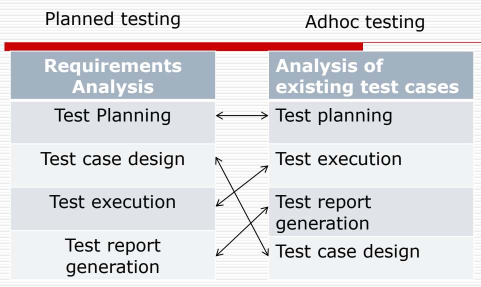
- Fake language testing
- ensures that switching between languages works properly and correct messages are picked up from proper directories that have the translated messages.

	Hello !	- English
Hello !	Ellohay !	- Pig Latin
	Hello !	- Wide Roman

- Language Testing
  - Check the functionality on English, one non-English, and one double-byte language platform combination
  - Check the performance of key functionality on different language platforms.
- Localization Testing

  It is to ensure whether the software functions as expected in the localized environment. This is called localization.

- ADHOC Testing
  - Adhoc testing are carried out in an unplanned manner.
  - Some of the issues faced by planned testing are as follows:
    - Lack of clarity in requirements and other specifications
    - Lack of skills for doing the testing
    - Lack of time for test design
  - Adhoc testing attempts to bridge the above gaps. It is done to explore undiscovered areas I the product by using intuition, previous experience of testing similar product.
  - Adhoc testing does not make use of any of the test case design techniques like EC partitioning, BV analysis and so on.
  - Fig shows the various types of adhoc testing and illustrates the basic difference between adhoc testing and planned testing.



- The fundamental difference between planned testing and adhoc testing is that the test execution and test report generation takes place before test case design in adhoc testing
- Since adhoc testing brings in new perspectives it is necessary to know what is already covered in the existing planned testing activities and also what changes in the product can cause changes in the testing functions
- Adhoc testing can be planned in one of two ways
  - After a certain number of planned test cases are executed. In this case, the product is likely to be in a better shape
  - Prior to planned testing. This will gain better clarity on the requirements

- Adhoc testing may cause a tester to jump across different functionalities and different screens. This is called random sampling test. This is also called "monkey testing"
- There are different types of adhoc testing, they are
  - Buddy Testing
  - Pair Testing
  - Exploratory Testing
  - Iterative Testing
  - Agile and Extreme Testing
- Buddy Testing
  - This testing uses the Buddy system practice wherein two team members are identified as buddies The buddies mutually help each other with a common goal of identifying defects early and correcting them

- A developer and a tester usually become buddies. Buddying people with good working relationships yet having diverse background is a kind of a safety measure that improves chances of detecting errors in the program very early.
- They also should be made to appreciate that they have a responsibility to one another.
- The code is unit tested to ensure what it is supposed to do before buddy testing starts
- Buddy testing uses both white box and black box testing approaches

- Pair Testing
- This uses two testers to pair up to test a product's feature on the same machine. The objective of this to maximize the exchange of ideas between the two testers
- When one person is executing the tests, the other person takes notes.
- They can swap roles of tester and scribe during a pair with multiple persons during various points of time of testing
- Pair testing is usually a focused session for about an hour or two.
- In Pair testing, one senior member can also help in pairing
- Pair testing can be done during any phase of testing
- Pair testing can track those elusive defects that are not caught by a single person testing.

- Exploratory Testing
- Another technique to find defects in adhoc testing is to keep exploring the product, covering more depth and breadth. Exploratory testing can be done during any phase of testing
- Exploratory testers may execute their tests based on their past experiences in testing a similar product, or a product of similar domain.
- Exploratory testing techniques
  - Guesses – used to find the part of the program that is likely to have more errors.
  - Architecture diagrams and use cases. - Arch. Diagram depict the interactions and relationships between different components and modules. Use case give an insight of the product's usage from user's perspective.

- Study of past defects – studying the defects reported in the previous releases helps in understanding of the error prone functionality in a product development environment.
- Error handling – is a portion of the code which prints appropriate messages or provides appropriate actions in case of failure
- Discussion – planned based on the understanding of the system during project discussion or meetings. Plenty of information can be picked up during these meetings
- Questionnaires and checklists – perform the exploration. Questions like what ,when, why how and who can provide leads to explore areas in the product.

- Iterative Testing
  - Iterative testing is where the requirement keep coming and product is developed iteratively for each requirement. This testing associated for this process is called iterative testing.
  - When a new requirement or a defect fix is done, it may have an impact on other requirements that have already been tested.
  - Majority of these tests are conducted manually.
- Agile and Extreme Testing
  - Agile XP models take the processes to the extreme to ensure that customer requirements are met in a timely manner In this mode customers partner with the project teams to go step by step in bringing the project to completion

---

- Testing OO Systems
- Testing OO System should tightly integrate data and agorithms.
- Testing OO Systems broadly covers the following topics
  - Unit testing a class
  - Putting classes to work together
  - System testing
  - Regression testing
- Unit testing a class
  - As a class is built before it is "published" for use by others, it has to be tested to see if it is ready for use
  - Classes are the building blocks for an entire oo systems
  - So the classes have to be unit tested

- Why the classes have to be individually tested
  - A class is intneded for heavy reuse
  - Many defects get introduced at the time a class gets defined
  - A class may have different features. Different clients pickup different parts of the class
  - A class is a combination of data and methods
  - It puts special features like inheritance, puts more context into the building blocks
  - Every class has certain variables.
  - Since a class is meant to be instantiated multiple times by different clients, various techniques of stress testing to be concentrated.

- Integration Testing
  - An OO system is not a collection of discrete objects or classes but these objects or classes should coexist, integrate and communicate with another.
  - Since OO systems are designed to be made up of a number of smaller components or classes that are meant to be reused.
- System testing and Interoperability of OO Systems
  - A class may have different parts, not all of which are used at the same time. Different parts of a class used by different clients may introduce defects at a later
  - Different classes may be combined together by a client and this combination may lead to new defects.
  - An instantiated object may not free all its resources allocated to it, thus causing memory leaks.

- Regression Testing of OO Systems
  - As a result of heavy reliance of OO systems on reusable components, changes to any one component could have unintended side-effects on the clients that use the component.
  - Hence frequent integration and regression runs becomes very essential for testing OO systems.

Usability and Accessibility Testing

- Usability testing attempts to characterize the "look and feel" and usage aspects of a product, from user's point of view.
- Some of the important thing for the usability testing are :
  - Usability and look and feel aspects are subjective in nature and cannot always be objectively measured
  - Perceptions of "good usability" varies from user to user.
  - User interface can be constructed as a design time activity. If a particular user interface does not meet the user's needs, it is a problem with requirement gathering.
- Characteristics of usability testing
  - Usability testing tests the product from the user's point of view
  - It is for checking the product to see if it is easy to use from the various categories of users
  - It is a process to identify discrepancies between the user interface of the product and the human user requirements.

Approach to Usability

- When doing usability testing, human factors can be represented in a quantifiable way and can be tested objectively.
- The number of mouse clicks, number of sub-menus to navigate , number of keystrokes, number of commands to perform a task can all be measured and checked as a part of usability testing.
- Usability testing is not only for product binaries or executables. It also applies to documentation and other deliverables that are shipped along with a product.
- Generally the people best suited to perform usability testing are
  - Typical representatives of the actual user segments who would be using the product
  - People who are new to the product, so that they can start without any bias and be able to identify usability problems.

- Aspect of usability with respect to messages that a system gives to its users also to be tested.
- Messages are classified into three types – informational, warning, and error
- When there is an information message, it is verified to find out whether an end user can understand that message and associate with operation done
- When there is a warning, such a message is checked for why it happened and what to do to avoid warning
- Whenever there is an error message, three things are looked for – What is the error, why that error happened, and what to do to avoid or work around the error.

When to do Usability Testing

- Usability testing is performed in two phases
  - Design validation
  - Part of the component and integration testing
- Usability design is verified through several means. Some of them are as follows
  - Style sheets –
  - Screens prototypes
  - Paper designs
  - Layout designs

Client Application	Web Application
Design for functionality	Design for user interface
Perform coding for functionality	Perform coding for user interface
Design for user interface	Test user interface (Phase 1)
Perform coding for user interface	Design for functionality
Integrate user interface with functionality	Perform coding for functionality
Test the user interface along with functionality (Phase 1 and 2)	Test user interface along with functionality (Phase 2)

- How to achieve Usability ?
- User interface requirements cannot be expressed in terms of words.
- One has to use the product to get a feel of its usability
- Involving customers to give feedback on the user interface during design phase is essential but this is always not possible because
  - The users may not have the time to participate in this exercise
  - There may be a number of users who may give conflicting requirements and it becomes impossible to satisfy everyone
  - The users may not even be able to visualize the usage of the product. Hence feedback may not be directly relevant.
- There can be different categories of users. Some users can be experts some can be beginners, and some novices.

- Expert users do not report usability problems. Instead they adapt themselves to the product hoping for a better products
- Beginners generally get impacted because of lack of usability but still they do not report usability
- Novice users report plenty of usability issues
- Irrespective of the category of users, the product is expected to be usable by all.
- There is also an additional dimension to the category of users that even challenged users such as those who are hearing impaired, vision impaired and mobility impaired.
- This type of usability testing deals with overall, visual and motion and motion-challenged users is called accessibility testing.

- In order to create to a product that is usable for all categories of the users, at least one user from each category must be involved in usability testing.
- When the users are using the product during usability testing, their activities are closely monitored and all observations are recorded and defects are raised by the test team rather than problem to be reported
- Recording the operating sequence, screens, and user reactions and making observations needs some equipment to be set up for a usability lab.

- When making observations certain guidelines are created and verified during usability testing Some of the items in the checklist are as follows.
  - Do users complete the assigned tasks /operation successfully
  - If so, how much time do they take to complete the task
  - Is the response from the product fast enough to satisfy them ?
  - Where did the users get struck? What problems do they have?
  - Where do they get confused ? Were they able to continue on their own? What helped them to continue?

- Quality factors for Usability
- Some quality factors are important when performing usability testing.
- Comprehensibility
  - Product should have simple and logical structure of features and documentation They should be grouped on the basis of user scenarios and usage When features and components are grouped in a product, they should be based on user terminologies
- Consistency
  - A Product needs to be consistent with any applicable standards, platform look-and-feel, base infrastructure

and earlier versions of the same product

- Navigation
  - The number of mouse clicks, or menu navigations that is required to perform an operation should be minimized to improve usability.
- Responsiveness
  - How fast the product responds to the user request . This should not be confused with the performance testing.
- Aesthetics Testing
  - Something that is acceptable to one person may appear ugly to another person
  - Adequate care, for the aesthetics aspect of the product can ensure that product is beautiful, at least a product must end up being termed ugly.

- A pleasant look for menus, pleasing colors, nice icons, and so on can improve aesthetics
- Aesthetics testing can be performed by anyone who appreciates beauty. Involving them during design testing phases and incorporating their inputs may improve the aesthetics of the product.

Accessibility Testing

- There are large number of people who are challenged with vision, hearing and mobility related problems. Product usability does not look into their requirements would result in lack of acceptance.
- For such users, alternative methods of using the product have to be provided.
- There are several tools available to help them with alternatives
- These tools are generally referred as accessibility tools or assistive technologies.
- Part of usability test planning

- Accessibility to the product can be provided by two means
  - Making use of accessibility features provided by the underlying infrastructure (operating sys.) called the accessibility and
  - Providing accessibility in the product through standards and guidelines called product accessibility
- Basic Accessibility
  - Basic accessibility is provided by the hardware and operating system. All the input and output devices of the computer and their accessibility options are categorized under basic accessibility.

- Keyboard Accessibility – is the most complete device for vision and mobility impaired users. Little projection helps vision impaired users to get a feel and align their fingers for typing.
  - Sticky keys
  - Filter keys – helps stopping the repetition completely or slowing down the repetition
  - Toggle key sound
  - Sound keys – help vision impaired users. -pronounces each character as and when they are hit
  - Arrow keys and control mouse – mobility impaired users – users can use keyboard arrow keys for mouse movements.

Screen Accessibility

- Many of the keyboard accessibility features assist the vision-impaired and mobility impaired users. These features may not help the hearing –impaired users who require extra visual feedback on the screen. Some accessibility features that enhance usability using the screen are as follows
  - Visual sound – wave form or graph form of the sound . These effects inform the user of the events that happen on the system using the screen
  - Enabling captions for multimedia – Speech and sound be enabled with text equivalents
  - Soft keyboard – users by displaying the keyboard on the screen.
  - Easy reading with high contrast – this mode helps pleasing colors and font sizes for all the menus on the screen

- Other accessibility Features
- There are many other accessibility features provided at the operating system level. A vision or mobility impaired user can find both keyboard and mouse devices difficult to use. In such cases, the option to use any other device should be provided.
- For example, a joystick device can be used as an alternative to a printing device and such a pointing device can be used in combination with a soft keyboard to accomplish the thigs a user desires.

Defect Bash Elimination

- Defect bash is an adhoc testing where people performing different roles in an organization test the product together at the same time
- The testing by all the participants during defect bash is not based on written test cases. What is to be tested is left to an individual's decision and creativity.
- Defect bash brings together plenty of good practices that are popular in testing industry They are as follows

- 1. Enabling people – " *Cross boundaries and test beyond assigned areas* "
- 2. Bringing different people performing different roles together in the organization for testing – " *Testing isn't for testers alone"*
- 3. Letting everyone in the organization use the product before delivery – " *Eat your own dog food"*
- 4. Bringing fresh pairs of eyes to uncover new defects – " *Fresh eyes have less bias"*
- 5. Bringing in people who have different levels of product understanding to test the product together randomly – " *Users of software are not same"*
- 6. Let testing doesn't wait for lack of time/taken for documentation – *"Does testing wait till all documentation is done ?"*

- 7. Enabling people to say *"system works" as well as enabling them to "break the system"* – *"Testing isn't to conclude the system works or doesn't work* "
- Even though it is said that defect bash is an adhoc testing, not all activities of defect bash are unplanned. All the activities in the defect bash are planned activities, except for what to be tested. It has several steps
  - Step 1: Choosing the frequency and duration of defect bash
  - Step 2 : Selecting the right product build
  - Step 3 : Communicating the objective of each defect bash to everyone
  - Step 4 : Setting up and monitoring the lab for defect bash
  - Step 5 : Taking actions and fixing issues
  - Step 6 : Optimizing the effort involved in defect bash

- Choosing the Frequency and duration of Defect bash

  It involves large amount of effort and involves huge planning. Frequent defect bashes will incur low return on investment, and too few defect bashes may not meet the objective of finding all defects.

- Selecting the Right product Build

  Since it involves large no. of people, effort and planning, a good quality build is needed for defect bash. A regression tested build would be ideal as all new features and defect fixes would have been already tested in such a build.

- Communicating the objective of Defect bash

  As Defect bash involves people performing different roles, the contribution they make has to be focused towards meeting the purpose and objective of defect bash. The objective should be to find a large number of uncovered defects or finding out system requirements or random defects . Defects that a test engineer finds easily should not be the objective of a defect bash

- Setting up and Monitoring the Lab

  Since defect bashes are planned, we need to setup and monitor a lab for this purpose. Finding out the right configuration, resources are activities that have to be planned carefully before a bash actually starts. The majority of defect bash fail due to inadequate hardware, wrong software configurations and perceptions related to performance and scalability of the software.

  There are two types of defects that will emerge during defect bash. The defects that are in the product, as reported by the users are functional defects.

  The defects related to system resources, such as memory leak, long turnaround time, missed requests, impact and utilization of system resources and so on are called non-functional defects.

- Optimizing the effort involved in Defect bash

  If a record of objectives and outcome is kept, having a tested build, keeping the right setup, sharing the objectives and so on, will save effort and meet the purpose of the objectives.

  Another approach to reduce the defect bash effort is to conduct "micro level" defect bashes before conducting one on a large scale. Defect bash can be further classified into

  1. Feature / component defect bash
  2. Integration defect bash
  3. Product defect bash

Acceptance Testing

- Acceptance Testing is a phase after system testing that is normally done by the customers or representatives of the customer
- Sometimes acceptance test cases are developed jointly by the customers and product organization
- Black box type of test cases.
- Apart from verifying the functional requirements, acceptance tests are run to verify the non-functional aspects of the system also.
- Acceptance test cases failing in a customer site may cause the product to be rejected

- Acceptance Criteria
  - Acceptance criteria – Product acceptance
    - Each requirement is associated with acceptance criteria. It is possible that one or more requirements may be mapped to form acceptance criteria.
    - This is not meant for executing test cases that have not been executed before
    - Testing for adherence to any specific legal or contractual terms is included in the acceptance criteria.
  - Acceptance criteira – Procedure acceptance
    - Based on procedures followed for delivery Ex. Documentation and release
      - User, administration and troubleshooting documentation should be part of the release.

    - Along with binary code, the source code of the product with build scripts to be delivered in a CD
    - A minimum of 20 employees are trained on the product usage prior to deployment
  - These procedural acceptance criteria are verified/ tested as part of acceptance testing.

- Acceptance criteria – service level agreements
  - Service level agreements are generally part of a contract signed by the customer and product organization
  - For ex : time limits to resolve those defects can be mentioned part of SLA such as
    - Major defects come up during first three months need to be fixed free of cost
    - Downtime of the system should be less than 0.1%
    - Major defects are to be fixed within 48 hours of reporting

- Selecting test cases for Acceptance testing
  - End-to-end functionality verification – Business transactions are tested as a whole and completed successfully
  - Domain tests – acceptance tests focus on business scenarios, the product domain test are also included.
  - Use scenario tests – real-life user scenario verification is to be tested
  - Basic sanity tests – Basic existing behavior of the product are tested.
  - New functionality – changes – test should focus new features.
  - Few non-functional tests – non functional tests are included and executed as part of acceptance testing

- Tests pertaining to legal obligations –
- Acceptance test data