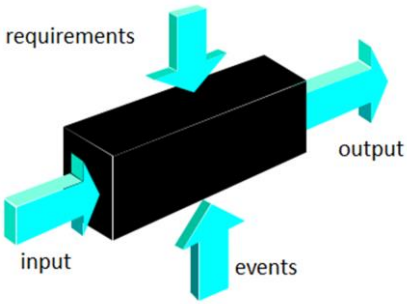
Black box testing:

It is focus on the functional requirements of the software. Black box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black box testing is not an alternative to white box techniques. It is a complementary approach that is likely to uncover a different class of errors than white box methods.

Black box testing attempts to find errors in the following categories:



(1) Incorrect or missing functions.
(2) Interface errors.
(3) Errors in data structures or external database access.
(4) Performance errors.
(5) Initialization and termination errors.

Unlike white box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing.

Why Black-Box Testing:

1. How is functional validity tested?
2. How is system behavior and performance tested?
3. What classes of input will make good test cases?
4. Is the system particularly sensitive to certain input values?
5. How are the boundaries of a data class isolated?
6. What data rates and data volume can the system tolerate?
7. What effect will specific combinations of data have on system operation?

Black box testing techniques:

- Equivalence Class Partitioning
- Boundary Value Analysis

Equivalence Class Partitioning:

Equivalence class testing is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behaviour.

The partitioning is done such that the program behaves in a similar way to every input value belonging to an equivalence class. Test cases are designed to test the input or output domain partitions. Equivalence class is determined by examining and analysing the input data range. Only one test case from each partition is required, which reduces the number of test cases necessary to achieve functional coverage.

The success of this approach depends upon the tester being able to identify partitions of the input and output spaces for which, in reality, cause distinct sequences of program source code to be executed.

"A test case should invoke as many different input considerations as possible to minimize the total number of test cases necessary."

"You should try to partition the input domain … into a finite number of equivalence classes such that you can reasonably assume … that a test of a representative value of each class is equivalent to a test of any other value."

Boundary Value Analysis:

Boundary value analysis is performed by creating tests that exercise the edges of the input and output classes identified in the specification. Test cases can be derived from the 'boundaries' of equivalence classes.

Typically programming errors occur at the boundaries of equivalence classes are known as "Boundary Value Analysis". Generally some time programmers fail to check special processing required especially at boundaries of equivalence classes. A general example is programmers may improperly use < instead of <=. The choices of boundary values include above, below and on the boundary of the class.

Types of Testing:

Unit (Module) Testing: Testing of a single module in an isolated environment

Integration testing: Testing parts of the system by combining the modules

Validation testing: Testing the functions of software.

System testing: Testing of the system as a whole after the integration phase

Acceptance testing: Testing the system as a whole to find out if it satisfies the requirements specifications.

1. Unit testing:

Unit testing, focuses verification effort on the smallest unit of software design—the module. Using the detail design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors detected as a result is limited by the constrained scope established for unit testing. The unit test is always white box-oriented. And the step can be conducted in parallel for multiple modules. For unit testing we need to isolate the module we want to test, we do this using two things, drivers and stubs.

Driver: A program that calls the interface procedures of the module being tested and reports the results. A driver simulates a module that calls the module currently being tested.

Stub: A program that has the same interface as a module that is being used by the module being tested, but is simpler. A stub simulates a module called by the module currently being tested.

2. Integration Testing:

Once all modules have been unit tested. If they all work individually, "why do you doubt that they'll work when we put them together?"

The problem of putting them together: "Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels global data structures can present problems".

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.

The objective is to take unit-tested modules and build a program structure that has been dictated by design. There is often a tendency to attempt ***non-incremental integration***; that is to construct program using a **"big bang"** approach where all modules are combined in advance. The entire program is tested as a whole. And chaos usually results. A set of errors are encountered. Correction is difficult because the isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected. New ones appear and the process continues in a seeming endless sloop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small segments. Where errors are easier to isolate and correct, interfaces are more likely to be tested completely and a systematic test approach may be applied.

- **Top-Down Integration**

Top-down integration is an incremental approach to the construction of program structure. Modules are integrated by moving downward through the control hierarchy. Beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a *depth -first* or *breadth-first* manner.

Depth-first integration would integrate all modules on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, modules M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all modules directly subordinate at each level, moving across the structure horizontally. From the figure, modules M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level; M5, M6, etc., follows.

Integration Process:

The integration process is performed in a series of five steps:

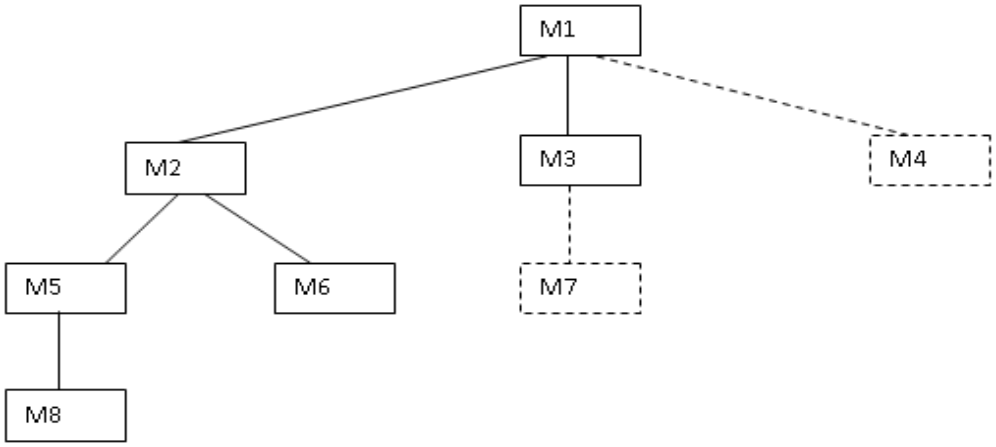
1. The main control module is used as a test driver and stubs are substituted for all modules directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e.. depth- or breadth-first), subordinate stubs are replaced one at a time with actual modules.
3. Tests are conducted as each module is integrated.

4. On the completion of each set of tests, another stub is replaced with the real module.
5. Regression testing (i.e.. conducting all or some of the previous tests) may be conducted to ensure that new errors have not been introduced.

The process continues front step 2 until the entire program structure is built.

Example: Integration Process:

Figure bellow illustrates the process. Assuming, a depth-first approach and a partially completed structure. Stub S7, is the next to be replaced with module M7. M7 may itself have stubs that will be replaced with corresponding modules. It is important to note that at each replacement tests are conducted to verify the interface.

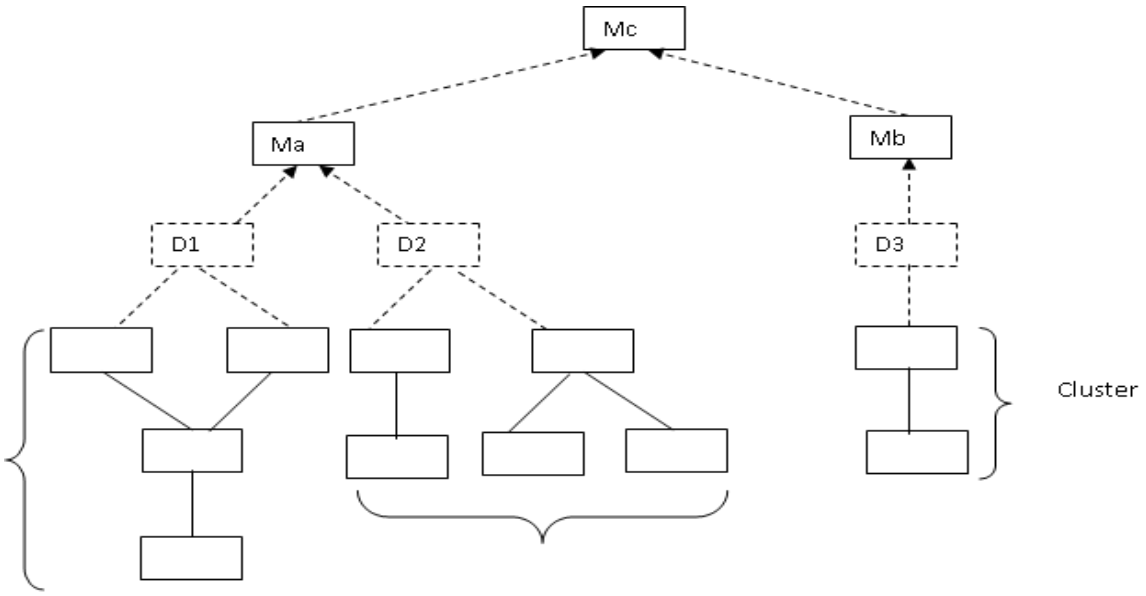


- **Bottom-up integration testing**

As its name implies, begins construction and testing with *atomic modules* (i.e.. modules at the lowest levels in the program structure). Because modules are integrated from the bottom up, processing required for modules subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level modules are combined into *clusters* (sometimes called *builds*) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.



Comments on Integration Testing:

The major disadvantage of the top-down approach is the need for *stubs* and the attendant testing difficulties that can be associated with them. Problems associated with stubs may be offset by the disadvantage of testing major control functions early. The disadvantage of the bottom-up approach is that "the program as an entity does not exist until the last module is added". This drawback is tempered by easier test case design and lack of stubs.

The selection of an integration strategy depends upon software characteristics and sometimes project schedule. In general a combined approach called ***sandwich testing*** that uses the top-down strategy for the upper level of program structure, coupled with a bottom-up strategy for the subordinate levels.

3. Validation Testing:

The validation testing succeeds when the software functions in manner that can be reasonably expected by the customer. It is achieved through a series of black box tests that demonstrate conformity with requirements. After each validation test case has been conducted, one of two possible conditions exists:

(1) The function or performance characteristics conform to specification and are accepted, or

(2) a deviation from specification is uncovered and a deficiency list is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled

completion. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

4. Acceptance Testing:

Acceptance Testing means testing the system as a whole to find out if it satisfies the requirements specifications. If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called ***alpha*** **and** ***beta*** *testing* to uncover errors that only the end user seems able to find.

When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements. Conducted by the end user rather than the system developer, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests.

The *alpha test* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more customer sites by the end user of the software. Unlike alpha testing, the developer generally is not present. The customer records all the problems that are encountered during beta testing and reports these to developer at regular intervals. As a result of problems reported during beta test, the software developer makes modifications and then prepares for release of the software product to the entire customer base.

5. System testing :

The software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., new hardware, information), and a series of system integration and validation tests are conducted. These tests Fall outside the scope of the software engineering process and are not conducted solely by the software developer. A classic system testing problem is "***finger pointing***". This occurs when a defect is uncovered. And one system element developer blames another for the problem.

The software engineer should anticipate potential interfacing problems:

(1) Design error- handling paths that test all information coming from other elements of the system.

(2) conduct a series of tests that simulate bad data or other potential errors at the software interface,

(3) Record the results of tests to use as "evidence" if finger pointing does occur

(4) Participate in the planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work should verify that all system elements have been properly integrated and perform allocated functions.