BEST PRACTICE
Validation Techniques

Validation assures that the end product (system) meets requirements and expectations under defined operating conditions. Within an IT environment, the end product is typically executable code. Validation ensures that the system operates according to plan by executing the system functions through a series of tests that can be observed and evaluated for compliance with expected results.

Table 7-4 illustrates how various techniques can be used throughout the standard test stages. Each technique is described below.

Table 7-4 Validation Techniques Used in Test Stages

Techniques / Test Stages	White-box	Black-box	Incremental	Thread	Regression
Unit Test	X				X
String/Integration Test	X	X	X	X	X
System Test		X	X	X	X
Acceptance Test		X			X

White-Box

White-box testing (logic driven) assumes that the path of logic in a unit or program is known. White-box testing consists of testing paths, branch by branch, to produce predictable results. Multiple white-box testing techniques are listed below. These techniques can be combined as appropriate for the application, but should be limited, as too many techniques can lead to an unmanageable number of test cases.

Statement Coverage	Execute all statements at least once.
Decision Coverage	Execute each decision direction at least once.
Condition Coverage	Execute each decision with all possible outcomes at least once.
Decision/Condition Coverage	Execute all possible combinations of condition outcomes in each decision, treating all iterations as two-way conditions exercising the loop zero times and once.
Multiple Condition Coverage	Invoke each point of entry at least once.

When evaluating the paybacks received from various test techniques, white-box or program-based testing produces a higher defect yield than the other dynamic techniques when planned and executed correctly.

Black-Box

In black-box testing (data or condition driven), the focus is on evaluating the function of a program or application against its currently approved specifications. Specifically, this technique determines whether combinations of inputs and operations produce expected results. As a result, the initial conditions and input data are critical for black-box test cases.

Three successful techniques for managing the amount of input data required include:

Equivalence Partitioning

An equivalence class is a subset of data that represents a larger class. Equivalence partitioning is a technique for testing equivalence classes rather than undertaking exhaustive testing of each value of the larger class. For example, a program which edits credit limits within a given range (at least $10,000 but less than $15,000) would have three equivalence classes:

- Less than $10,000 (invalid)
- Equal to $10,000 but not as great as $15,000 (valid)
- $15,000 or greater (invalid)

Boundary Analysis

This technique consists of developing test cases and data that focus on the input and output boundaries of a given function. In the credit limit example, boundary analysis would test the:

- Low boundary plus or minus one ($9,999 and $10,001)
- Boundaries ($10,000 and $15,000)
- Upper boundary plus or minus one ($14,999 and $15,001)

Error Guessing

This is based on the theory that test cases can be developed from the intuition and experience of the tester. For example, in a test where one of the inputs is the date, a tester may try February 29, 2000 or February 29, 2001.

Incremental

Incremental testing is a disciplined method of testing the interfaces between unit-tested programs and between system components. It involves adding unit-tested programs to a given module or component one by one, and testing each resultant combination. There are two types of incremental testing:

Top-Down

> This method of testing begins testing from the top of the module hierarchy and works down to the bottom using interim stubs to simulate lower interfacing modules or programs. Modules are added in descending hierarchical order.

Bottom-Up

> This method of testing begins testing from the bottom of the hierarchy and works up to the top. Modules are added in ascending hierarchical order. Bottom-up testing requires the development of driver modules, which provide the test input, call the module or program being tested, and display test output.

There are pros and cons associated with each of these methods, although bottom-up testing is generally considered easier to use. Drivers tend to be less difficult to create than stubs, and can serve multiple purposes. Output from bottom-up testing is also often easier to examine, as it always comes from the module directly above the module under test.

Thread

This test technique, which is often used during early integration testing, demonstrates key functional capabilities by testing a string of units that accomplish a specific function in the application. Thread testing and incremental testing are usually used together. For example, units can undergo incremental testing until enough units are integrated and a single business function can be performed, threading through the integrated components.

When testing client/server applications, these techniques are extremely critical. An example of an effective strategy for a simple two-tier client/server application could include:

1. Unit and bottom-up incrementally test the application server components.
2. Unit and incrementally test the GUI or client components.
3. Test the network.

4. Thread test a valid business transaction through the integrated client, server, and network.

Regression

There are always risks associated with introducing change to an application. To reduce this risk, regression testing should be conducted during all stages of testing after a functional change, reduction, improvement, or repair has been made. This technique assures that the change will not Regression testing can be a very expensive undertaking, both in terms of time and money. The test manager's objective is to maximize the benefits of the regression test while minimizing the time and effort required for executing the test.

The test manager must choose which type of regression test minimizes the impact to the project schedule when changes are made, and still assures that no new defects were introduced. The types of regression tests include:

Unit Regression Testing

This retests a single program or component after a change has been made. At a minimum, the developer should always execute unit regression testing when a change is made.

Regional Regression Testing

This retests modules connected to the program or component that have been changed. If accurate system models or system documentation are available, it is possible to use them to identify system components adjacent to the changed components, and define the appropriate set of test cases to be executed. A regional regression test executes a subset of the full set of application test cases. This is a significant timesaving over executing a full regression test, and still helps assure the project team and users that no new defects were introduced.

Full Regression Testing

This retests the entire application after a change has been made. A full regression test is usually executed when multiple changes have been made to critical components of the application. This is the full set of test cases defined for the application.

When an application feeds data to another application, called the "downstream" application, a determination must be made whether regression testing should be conducted with the integrated application.

Testers from both project teams cooperate to execute this integrated test, which involves passing data from the changed application to the downstream application, and then executing a set of test cases for the receiving application to assure that it was not adversely affected by the changes.

References

Guide – CSTE Common Body Of Knowledge, V6.1