**6. Software Testing**

Testing is a set of activities that can be planned in advanced and conducted systematically, then a template for software testing should be defined for the software process. Software testing is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function.

Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

**6.1 Unit Testing**

Unit testing focuses verification effort on the smallest unit of software design; the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.

The tests that occur as part of unit tests are illustrated schematically in figure 6.1. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all errors handling paths are tested.

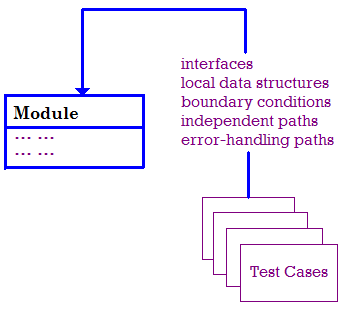


Figure 6.1 Unit Test

Tests if data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained during unit testing. Among the more common errors in computation are:

1. Misunderstood or incorrect arithmetic procedure.
2. Mixed mode operations.
3. Incorrect initialization.
4. Precision inaccuracy.
5. Incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another. Test cases should uncover errors such as:

1. Comparison of different data types.
2. Incorrect logical operators or precedence.
3. Expectation of quality when precision error makes equality unlikely.
4. Incorrect comparison of variables.
5. Failure to exit.
6. Improperly modified loop variables.

**6.2 Integration Testing**

Integration testing is a systematic technique for constructing program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design. There are number of different integration testing strategies.

**6.2.1 Top-Down Integration**

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

Referring to figure 28, depth-first integration would integrate all components on a major control path of the structure. Selection of a major control path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components Ml, M2, M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the right-hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3 and M4 would be integrated first. The next control level, M5, M6 and so on.



Figure 6.2: Top-Down Integration

**6.2.2 Bottom-Up Integration**

Bottom-up integration testing begins construction and testing with atomic modules (components at the lowest level in the program structure). Because components integrated from the bottom up, processing required for components subordinate to a given level is always available.

Integration follows the pattern illustrated in figure 6.3. Components are combined to form clusters 1, 2 and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc and so on:



Figure 6.3: Bottom-Up Integration