**SYSTEM TESTING**

**TEST PLAN**

The importance of software testing and its implications cannot be overemphasized. Software testing is a critical element of Software Quality Assurance and represents the ultimate review of the specifications, design and coding.

**TESTING OBJECTIVES**

The main objective of testing is to uncover a host of errors, systematically and with minimum effort and time. Stating formally, we can say, Testing is a process of executing a program with the intent of finding an error. A successful test is one that uncovers an as yet undiscovered error. A good test case is one that has a high probability of finding error, if it exists.

But there is one thing that testing cannot do (just to quote a very famous sentence) “Testing cannot show the absence of defects, it can only show that software defects are presents.”

As the test results are gathered and evaluated they begin to give a qualitative indication of the reliability of the software. If severe errors are detected, the overall quality of the software is a natural suspect. If, on the other hand, all the errors, which are encountered, are easily modifiable, then one of the two conclusions can be made:

* The tests are inadequate to detect possibly present errors.
* The software more or less confirms to the quality and reliable standards.

For the purpose of the current project we are assuming that in the event that errors that are easily modifiable points to the latter possibility, since repeating the entire testing routine can be very time consuming. What we propose to do instead is to get it tested by one or more persons who are not a part of the development team but is well versed with the subject and with the concept of software testing (alpha testing). If he can detect no serious errors, it will enable us to state with more confidence that the software does actually conform to the expected standards.

**TESTING STRATEGY**

A testing strategy is a roadway, giving there how-to conducting a test. Our testing strategy is flexible enough to promote customization that may be necessary in due course of development process. For instance during coding we find that a change in design ( e.g. Z demoralized table makes certain query easy to process), we maintain a change log and refer to it at appropriate time during the testing. Software can be tested in one of the following ways:

Knowing the specific functions that the Software is expected to perform, tests can be conducted to perform that all functions are fully operational. Knowing the internal workings of the product, tests can be conducted to show that internal operations of the system perform according to the specifications and all internal components are adequately exercised.

The first approach is what is known as Black box testing and the second is called White box testing. We will be using a mixed approach, more popularly known as sandwich testing. We apply white box testing techniques to ascertain the functionalities top-down and then we use black box testing to demonstrate that everything runs as expected.

**UNIT TESTING**

Unit testing focuses verification effort on the smallest unit of software i.e. the module. Using the detailed design and the process specifications testing is done to uncover errors within the boundary of the module. All nodules must be successful in the unit test before the start of the integration testing begins.

**INTEGRATION TESTING**

Integration testing is a systematic technique for constructing the program structure while conducting tests at the same time to uncover errors associated with interfacing. We have used incremental integration testing for this project.

**VALIDATION TESTING**

At the culmination of integration testing the software is complete as a package and the interfacing errors have been uncovered and fixed, final tests- validation testing- may begin. Validation tests succeed when the software performs exactly in the manner as expected by the user.

Software validation is done by a series of Black box tests that demonstrate the conformance with requirements. Alpha and beta testing fall in this category. We will not do beta testing but alpha testing will certainly will certainly be done.

**TEST CASE DESIGN**

The techniques that are used in deriving the test cases are explained below.

**Condition Testing**

Condition Testing is a test case design method that exercises the logical conditions contained in the program. The possible components in a condition statement are:

A Boolean operator, a Boolean variable, a relational operator, arithmetic expression and parenthesis around simple or compound conditions. The condition testing method focuses on testing each condition in the program.

It has two advantages over the pure white box testing:

* Measurement of test coverage is straightforward.
* The test coverage of conditions in a program provides guidance for generation of additional tests for the program.
* We have used a special condition testing method called the Domain Testing where the condition in a form lie E1 < Relational Operator> E2 will the tested for three conditions E1 > E2, E1 equal to E2 and E1 les than E2.

**Boundary Value Analysis**

Boundary value analysis leads to a selection of test cases that exercise the boundary conditions or bounding values. It has been observed that a large number of errors tend to appear at the boundaries of the input domain than in the center. The guidelines for developing the test cases in BVA are given below:

If the input condition has a low and high range then tests should be done at the low and high boundaries. Values above and below these extremes should be tested.

Apply the same principle to output conditions. E.g.: Test cases should be designed to generate the maximum and minimum no of entries in a report that is generated by the program.

**Equivalence Partitioning**

Equivalence partitioning is a black box testing method that divides the input domain of a program into classes of data from which test cases can be derived. A typical test case uncovers a class of errors that might otherwise require many more test cases before the error is observed.

Equivalence classes for input determine the valid and invalid inputs for the program. Equivalence class test cases are generated using the following guidelines:

* If an input class specifies a range then one valid and two invalid equivalence classes are defined.
* If an input class specifies a value then one valid and one invalid equivalence classes are defined.
* If an input class specifies a member of a set then one valid and one invalid equivalence classes are defined.
* If an input class specifies a Boolean then one valid and two invalid equivalence classes are defined.
* Test cases should be selected so that the largest number of attributes of an equivalence class is exercised at once.