Testing

Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction. Some commonly used terms associated with testing are:

Failure: This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

Test case: This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

Test suite: This is the set of all test cases with which a given software product is to be tested.

Aim of Testing

The aim of the testing process is to identify all defects existing in a software product. For most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Verification Vs Validation

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

Design of Test Cases

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite.

An optional test suite must be signed that is of reasonable size and can uncover as many errors existing in the system as possible.

If test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite.

Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing.

Testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered.

Design of Test Cases

Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

```
if (x>y)
max = x;
else
max = x;
```

For the above code segment, the test suite, *{(x=3,y=2);(x=2,y=3)}* can detect the error, whereas a larger test suite *{(x=3,y=2);(x=4,y=3);(x=5,y=1)}* does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

Design of Test Cases

Functional Testing Vs. Structural Testing

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing. On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

Black-Box Testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class partitioning
- Boundary value analysis

Equivalence Class Partitioning

- The domain of input values to a program is partitioned into a set of equivalence classes.
- This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.
- Testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class.
- Equivalence classes for a software can be designed by examining the input data and output data.

Equivalence Class Partitioning

The following are some general guidelines for designing the equivalence classes:

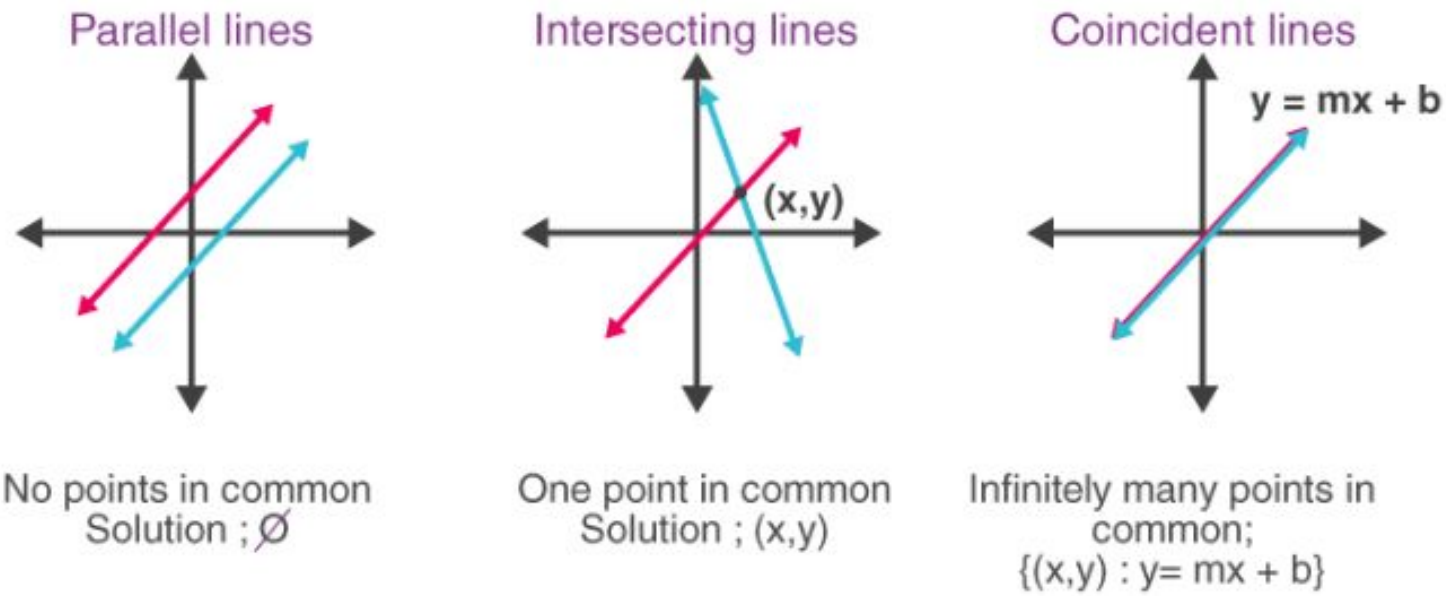
1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example 1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000.

Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Equivalence Class Partitioning

Example 2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.



The equivalence classes are the following: • Parallel lines (m1=m2, c1≠c2) • Intersecting lines (m1≠m2) • Coincident lines (m1=m2, c1=c2) Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

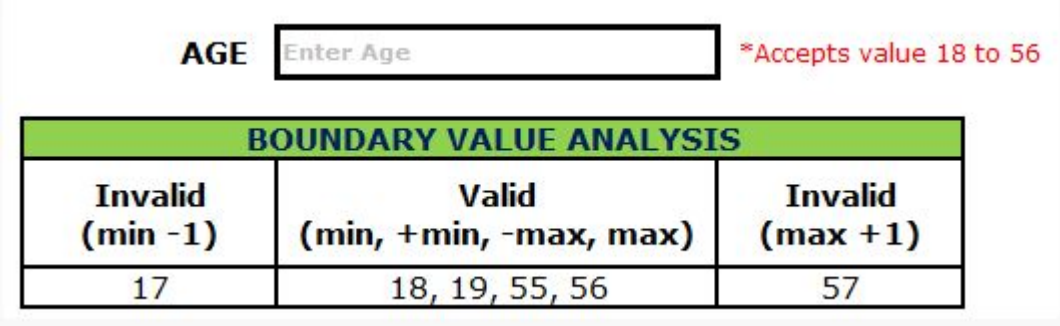
Boundary Value Analysis

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection test cases at the boundaries of the different equivalence classes.

Example 1: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1,5000,5001}.

Example 2: Assume, we have to test a field which accepts Age 18 – 56.

Minimum boundary value is 18
Maximum boundary value is 56
Valid Inputs: 18,19,55,56
Invalid Inputs: 17 and 57



BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	Valid (min, +min, -max, max)	Invalid (max +1)
17	18, 19, 55, 56	57