APPROACHES TO SOFTWARE TESTING

PROGRAM VERIFICATION AND VALIDATION

Validation: Are we building the right product? Does program meet expectations of user? Verification: Are we building the product right? Does program conform to its specifications?

Testing: The process of establishing the presence of faults.
Debugging: The process of finding and removing faults.
Bebugging: The process of estimating the number of bugs in a program
THE TESTING PROCESS
Unit testing - Individual components are tested to see if they work correctly. Each component is considered a stand-alone entity.

Module testing - A module is a collection of components which are interdependent. After each program unit has been tested, the interaction of these components when they interact must be tested. A module encapsulates related components and it should be possible to test a module as a stand-alone entity.

Subsystem testing - Modules are put together to form subsystems. Subsystems may be designed and implemented by different software engineers. The most common problems occur in the interfaces of subsystems.
Integration testing - Subsystems must be integrated to make up the entire system. Problems usually are caused by unanticipated interactions between subsystems and components.

Stress testing - Put an unnatural load on a system; perhaps exceed the number of transactions per second for the system.

Acceptance testing - Test the system with real data (sometimes called alpha testing).

Beta testing - Deliver product to a few customers or potential customers who agree to use the product and report errors.
TOP-DOWN AND BOTTOM-UP TESTING

Top-down testing involves starting at the subsystem level with modules represented by stubs - objects which have the same interface as the module but are much simpler. After all subsystems are tested in a similar fashion, the stubs are replaced by the actual code and testing continues. Top-down testing should be used with top-down program development so that a module is tested as soon as it is coded.

Bottom-up testing reverses this process. The components are tested individually; then components that make up a module are integrated and the module is tested. Finally, modules are integrated and the subsystem is tested.
Top-down testing has the advantage that a working (but limited) is available at an early stage of development. Provides a good psychological boost, and it demonstrates the feasibility of the system to management. Validation begins early in the development cycle. However, strict top-down testing can be difficult because program stubs that simulate lower levels of the system must be produced; also, realistic output may be difficult to produce.

If bottom-up testing is used, drivers must be constructed for the lower level modules which present these modules with appropriate inputs. Test cases are reasonably easy to generate. However, no demonstrable program is available until the very last module has been tested. Also, high level design errors may not be detected until the late stages of system test.
TESTING TECHNIQUES
In principle, testing of a program should be exhaustive. Every statement in the program should be exercised and every possible path combination through the program should be executed at least once. In practice, this is not reasonable in a program which contains loops as the number of paths is usually very large.
Some guidelines -
  1) Testing a system's capabilities is more important than testing its components. Test for errors that will stop the users from completing their jobs and not for irritations.
  2) Testing old capabilities is more important than testing new ones. Don't break a working system.
  3) Testing typical situations is more important than testing boundary value cases.
TEST CASES AND TEST DATA
Test cases and test data are not the same thing. Test data are the inputs that have been devised to test the system; test cases are input and output specifications plus a statement of the function under test.

Consider the testing of a simple routine to search a table of integers to determine if some given integer is present in that table. Assume the routine is called as

\[ S := \text{Search(AnArray, InValue)} \]
If AnArray has an element equal to InValue, the index of that element in AnArray is returned by Search, otherwise -1 is returned. Assume the programming language Search is written in checks the types of parameters and number of parameters. Also assume the intrinsic function FIRST and LAST return the upper and lower bounds of the array, and that the language (and operating system) will detect out-of-bounds array indices.

Test cases for Search(An Array, InValue):
(1) Array size of 1, element in array.
(2) Array size of 1, element not in array.
(3) Empty array.
(4) Even size array, element first element in array.
(5) Even size array, element last element in array.
(6) Even size array, element not in array.
(7) Odd size array, element first element in array.
(8) Odd size array, element last element in array.
(9) Odd size array, element not in array.
(10) Even size array, element in array, not first or last.
(11) Odd size array, element in array, not first or last.

Test cases for these situations:
  (1) Array is a single value equal to required value.
      Input: AnArray = 17; InValue = 17
      Output: function returns 1
  (2) Array is a single value not equal to required value.
      Input: AnArray = 17; InValue = 0
      Output: function returns -1
  ...
  ...
  (5) The array size is even and the last value is the required value
      Input: AnArray = 17, 18, 21, 23; InValue = 23
Output: function returns 4
...
...
(10) The array size is even and the element is neither the first nor last

Input: AnArray = 17, 23, 29, 35, 41, 45; InValue = 23
Output: function returns 2

This testing is called “black-box" testing because the tester is provided only with a description of the routine and does not have access to the routine.
The apparently arbitrary set of test cases were determined by the following heuristics:

(1) Search programs are most likely to go wrong when the key element is either the first or last element in the array.
(2) Programmers often don't consider situations where there are an unusual number of elements (zero or one) in the collection.
(3) Search routines often behave differently depending on whether the number of values in the array is even or odd.
EQUIVALENCE PARTITIONING

The form of input classification for determining the test inputs for the above search routine is called equivalence partitioning. If a program does not display an erroneous output for one member of a class, it should not do so for any member of that class. The equivalence class must be identified by using the program specification or user documentation and by the tester using experience to predict which classes of input value are likely to detect errors. For example, if a specification says that four to eight values are to be input, equivalence classes are less than four, between four and eight, and more than eight.

Output equivalence classes should also be considered and input values that generate outputs at the boundary of each output class should be chosen as test input. For example, say a program is
designed to produce between three and six outputs, with each output lying in the range 1000-2500. Test input should be selected which produces 3 values of 1000, 3 of 2500, 6 of 1000, and 6 of 2500. Furthermore, input should be selected so that erroneous output values would result if that input were processed as correct input. The input should attempt to force the program to produce less than three values, more than six values, values less than 1000 and values greater than 2500.

Suppose a procedure converts a string of digits to a 16-bit, two's complement integer. Each input can have up to 6 characters; if less than six characters, then blanks pad to the left. If an input is negative, a minus sign occupies the position immediately to the left of the most significant digit. Assume the input is a member of an array and that compiler checking will guarantee that no more than 6 characters will be input without generating an error. The
compiler will also check for any other invalid parameter types or values.
**Equivalence Classes**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Type</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Input</td>
<td>1-5 non-blank characters</td>
</tr>
<tr>
<td>C2</td>
<td>Input</td>
<td>6 non-blank characters</td>
</tr>
<tr>
<td>C3</td>
<td>Input</td>
<td>Empty</td>
</tr>
<tr>
<td>C4</td>
<td>Input</td>
<td>Single minus sign, no characters</td>
</tr>
<tr>
<td>C5</td>
<td>Input</td>
<td>Minus sign as most significant character</td>
</tr>
<tr>
<td>C6</td>
<td>Input</td>
<td>Digit as most significant character</td>
</tr>
<tr>
<td>C7</td>
<td>Input</td>
<td>Left padded with non-blank and not 0</td>
</tr>
<tr>
<td>C8</td>
<td>Input</td>
<td>Left padded with 0</td>
</tr>
<tr>
<td>C9</td>
<td>Input</td>
<td>Digit as significant character but with invalid characters in number</td>
</tr>
<tr>
<td>C10</td>
<td>Input</td>
<td>Gap between minus sign and number</td>
</tr>
<tr>
<td>C11</td>
<td>Output</td>
<td>Negative integers &gt;= Minint and &lt; 0</td>
</tr>
<tr>
<td>C12</td>
<td>Output</td>
<td>Zero</td>
</tr>
<tr>
<td>C13</td>
<td>Output</td>
<td>Positive integers &gt; 0 and &lt;= Maxint</td>
</tr>
</tbody>
</table>
### Test Cases

<table>
<thead>
<tr>
<th>Number</th>
<th>Input</th>
<th>Expected Output</th>
<th>Classes Tested.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bbbbb1</td>
<td>1</td>
<td>C1,C6,C13</td>
</tr>
<tr>
<td>2</td>
<td>000001</td>
<td>1</td>
<td>C2,C6,C8</td>
</tr>
<tr>
<td>3</td>
<td>bbb-1</td>
<td>-1</td>
<td>C5,C11</td>
</tr>
<tr>
<td>4</td>
<td>-00001</td>
<td>-1</td>
<td>C2,C5,C8</td>
</tr>
<tr>
<td>5</td>
<td>000000</td>
<td>0</td>
<td>C2,C12</td>
</tr>
<tr>
<td>6</td>
<td>bbbbb0</td>
<td>0</td>
<td>C6,C12</td>
</tr>
<tr>
<td>7</td>
<td>b32767</td>
<td>32767</td>
<td>C1,C6,C13</td>
</tr>
<tr>
<td>8</td>
<td>032767</td>
<td>32767</td>
<td>C2,C13</td>
</tr>
<tr>
<td>9</td>
<td>-32768</td>
<td>-32768</td>
<td>C2,C5,C11</td>
</tr>
<tr>
<td>10</td>
<td>b32768</td>
<td>Invalid Input</td>
<td>C1,C13</td>
</tr>
<tr>
<td>11</td>
<td>-32769</td>
<td>Invalid Input</td>
<td>C2,C5,C11</td>
</tr>
<tr>
<td>12</td>
<td>123456</td>
<td>Invalid Input</td>
<td>C2</td>
</tr>
<tr>
<td>13</td>
<td>xxxxx1</td>
<td>Invalid Input</td>
<td>C7</td>
</tr>
<tr>
<td>14</td>
<td>xxxx-1</td>
<td>Invalid Input</td>
<td>C7</td>
</tr>
<tr>
<td>15</td>
<td>bbbbbb</td>
<td>Invalid Input</td>
<td>C3</td>
</tr>
<tr>
<td>16</td>
<td>bbb2x1</td>
<td>Invalid Input</td>
<td>C9</td>
</tr>
<tr>
<td>17</td>
<td>bbb2-1</td>
<td>Invalid Input</td>
<td>C9</td>
</tr>
<tr>
<td>18</td>
<td>bbb-b1</td>
<td>Invalid Input</td>
<td>C10</td>
</tr>
<tr>
<td>19</td>
<td>bbbbb-</td>
<td>Invalid Input</td>
<td>C4</td>
</tr>
</tbody>
</table>
STRUCTURAL TESTING

Black-box testing is the name given to testing when the tester is presented with the specification of the component being tested and uses this to derive test cases. The advantage is the tester does not need source code and need not understand the program being tested. The disadvantage is that the tester cannot get clues from the program about which test inputs best exercise the program.

Another approach called white-box, glass-box, or structural testing relies on the tester having knowledge of the program to derive test data. The starting point for structural testing is to derive a program flow graph that shows all possible execution paths for a program so that test cases for each path can be designed.
The maximum number of independent paths in a program is equal to the program's cyclomatic number. An independent path is one which traverses at least one new edge in the flow graph. The tester must still discover a set of appropriate independent paths. This metric must be used carefully because it does not guarantee that the right test data is used or that a “good” set of independent paths are tested. It also doesn't address data driven programs well.

Consider a program represented by a flow chart and a more abstract version represented by a flow graph. The discussion and example on pages 189-196 of the Galin text demonstrates this transformation well.

The flow graph in that example is:
VG) = # Regions = 6

V(G) = Edges – Nodes + 2
= 21 - 17 + 2 = 6

V(G) = Binary Decisions + 1
= % + 1 = 6
TESTING REAL-TIME SYSTEMS

Real-time systems are systems where the processes comprising the system must respond to events under time constraints. The testing of real-time systems is particularly critical because the reliability requirements of these systems are usually greater than the requirements for systems which are not time critical. Real-time systems are normally made up from a number of distinct cooperating processes and they are often interrupt driven.

Real-time systems are especially difficult to test because of the subtle interactions that may arise among the various processes in the system. System errors may be time dependent, arising only when the system processes are in a particular state; that state may
be difficult to reproduce. These states normally arise because of parallel processes. One way to control integration and system testing is to identify “threads" of execution where an input is transformed by a number of processes in turn to produce an output. Once threads are tested, new event tests should be created to test for simultaneously occurring events
STATIC PROGRAM VERIFICATION (INCLUDES INSPECTIONS)

Static verification techniques do not require the program to be executed. Rather, they are concerned with examining the source code of a program and detecting faults in the code before execution. It has been reported that static verification can find 60% of the errors in a program before the program is executed.

Static program analysers are software tools that scan the source text of a program and detect possible faults and anomalies. They do not require that a program be executed. Some faults that can be checked are:
- Unreachable code
- Unconditional branches into loops
- Undeclared variables
- Parameter type mismatches
- Parameter number mismatches
- Uncalled functions and procedures
- Variables used before initiation
- Non-usage of function results
- Possible array bound violations
- Misuse of pointers