Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: **simple loops**, **concatenated loops**, **nested loops**, **and unstructured loops.**

Why do loop testing ?

1) It helps to identify loops initialization problems.
2) It helps to identify infinite loops.

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.

2. Only one pass through the loop.

3. Two passes through the loop.

4. *m* passes through the loop where $m < n$.

5. n _1, n, $n + 1$ passes through the loop.

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.

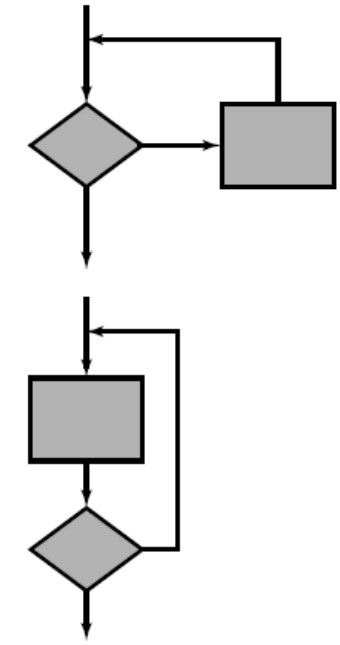
This would result in an impractical number of tests. An approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
4. Continue until all loops have been tested.

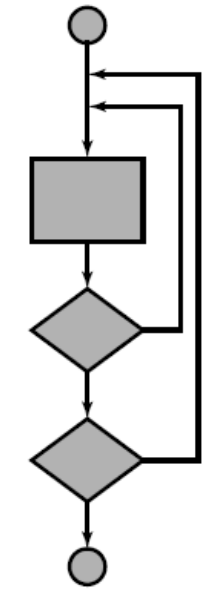
Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.

However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

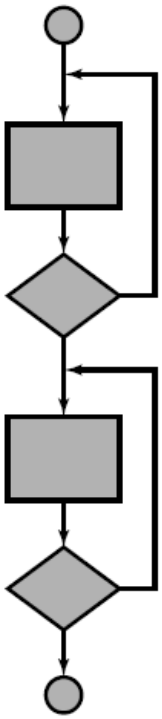
Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.



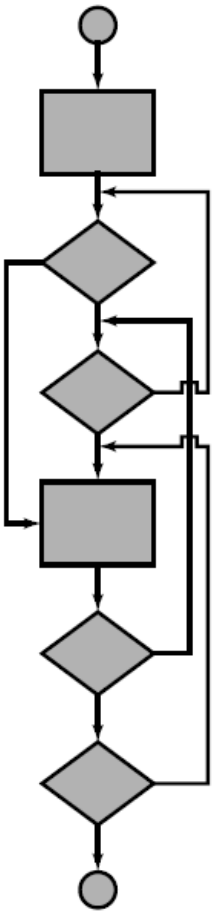
Simple loops



Nested loops



Concatenated loops



Unstructured loops

SOFTWARE TESTING STRATEGIES

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing should be defined for the software process.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

- Testing begins at the component and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and Validation

Software testing is one element of a broader topic that 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. In another way:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

The definition of V&V encompasses many of the activities that we have referred to as *software quality assurance* (SQA).

A Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure below. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding.

To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.

A strategy for software testing may also be viewed in the context of the spiral. *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter *validation testing*, where requirements established as part of software requirements analysis are validated against the software that has been constructed.

Finally, we arrive at *system testing*, where the software and other system elements are tested as a whole.

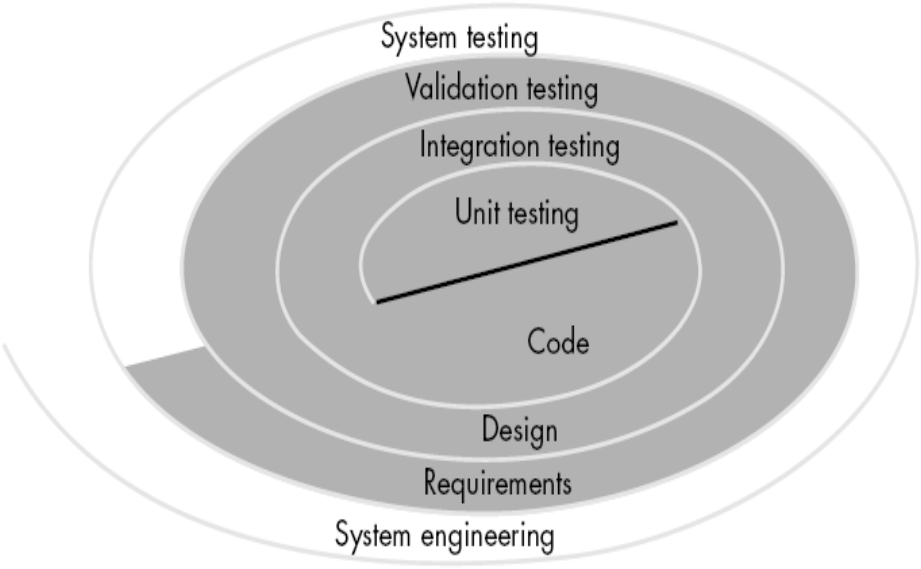


Figure. Testing Strategy