Software Quality Assurance

1. *Software Quality*

Developing quality products is critical for a company to be successful in the new global economy. Traditionally, efforts to improve quality have centered on the end of the product development cycle by emphasizing the detection and correction of defects.

On the contrary, the new approach to enhancing quality encompasses all phases of a product development process—from a requirements analysis to the final delivery of the product to the customer. Every step in the development process must be performed to the highest possible standard.

An effective quality process must focus on:

- Paying much attention to customer's requirements
- Making efforts to continuously improve quality
- Integrating measurement processes with product design and development
- Pushing the quality concept down to the lowest level of the organization
- Developing a system-level perspective with an emphasis on methodology and process
- Eliminating waste through continuous improvement

1.1 Defining Software Quality Assurance (SQA)

Software: is the combination of computer programs (the "code"), procedures, documentation, and data necessary for operating the software system. The combination of all four components is needed to assure the quality of the development process as well as the ensuing long years of maintenance.

Software Quality: is the degree of conformance to specific functional requirements, specified software quality standards, and Good Software Engineering Practices (GSEP).

Software Quality Assurance: is the systematic, planned set of actions necessary to provide adequate confidence that a software development or maintenance process conforms to established

functional technical requirements as well as the managerial requirements of keeping to schedules and operating within the budget.

Quality is a complex concept—it means different things to different people, and it is highly context dependent. In a comprehensive manner, Quality can be discussed in five views as follows:

1. **Transcendental View**: It pictures quality as something that can be recognized but is difficult to define. The transcendental view is not specific to software quality alone but has been applied in other complex areas.
2. **User View**: It perceives quality as fitness for purpose. According to this view, while evaluating the quality of a product, one must ask the key question: "Does the product satisfy user needs and expectations?"
3. **Manufacturing View**: Here quality is understood as conformance to the specification. The quality level of a product is determined by the extent to which the product meets its specifications.
4. **Product View**: In this case, quality is viewed as tied to the inherent characteristics of the product. A product's inherent characteristics, that is, internal qualities, determine its external qualities.
5. **Value-Based View**: Quality, in this perspective, depends on the amount a customer is willing to pay for it.

Various software quality models have been proposed to define quality and its related attributes. The most influential ones are the ISO 9126 and the CMM. The ISO 9126 quality model was developed by an expert group under the support of the International Organization for Standardization (ISO). The document ISO 9126 defines six broad, independent categories of quality characteristics:

1. Functionality,
2. Reliability,
3. Usability,
4. Efficiency,
5. Maintainability, and
6. Portability.

1.2 Software Errors, Software Faults and Software Failures

- **Software errors** are sections of the code that are partially or totally incorrect as a result of a grammatical, logical or other mistake made by a systems analyst, a programmer, or another member of the software development team.
- **Software faults** are software errors that cause the incorrect functioning of the software during a specific application.
- Software faults become **software failures** only when they are "activated", that is, when a user tries to apply the specific software section that is faulty. Thus, the root of any software failure is a software error.

1.3 Causes of Software Errors

There are nine causes of software errors:

1. faulty requirements definition,
2. client– developer communication failures,
3. deliberate deviations from software requirements,
4. logical design errors,
5. coding errors,
6. non-compliance with documentation and coding instructions,
7. short comings of the testing process,
8. procedure errors, and
9. documentation errors.

It should be emphasized that all causes of error are human, the work of systems analysts, programmers, software testers, documentation experts, and even clients and their representatives.

1.4 Objectives of Software Quality Assurance Activities

The objectives of SQA activities for software development are:

1. Assuring, with acceptable levels of confidence, conformance to functional technical requirements.
2. Assuring, with acceptable levels of confidence, conformance to managerial requirements of scheduling and budgets.

3. Initiating and managing activities for the improvement and greater efficiency of software development and SQA activities.

1.5 Software Quality Assurance and Quality Control

Quality control is a set of activities carried out with the main objective of withholding products from shipment if they do not qualify. In contrast, quality assurance is meant to minimize the costs of quality by introducing a variety of activities throughout the development and maintenance process in order to prevent the causes of errors, detect them, and correct them in the early stages of development. As a result, quality assurance substantially reduces the rates of non-qualifying products.

1.6 Software Quality Assurance and Software Engineering:

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software. The characteristics of software engineering, especially its systematic, disciplined and quantitative approach, make software engineering a good environment for achieving SQA objectives. It is commonly accepted that cooperation between software engineers and the SQA team is the way to achieve efficient and economic development and maintenance activities that, at the same time, assure the quality of the products of these activities.

1.7 Role of Testing in SQA

Testing plays an important role in achieving and assessing the quality of a software product. On the one hand, we improve the quality of the products as we repeat a test–find defects–fix cycle during development. On the other hand, we assess how good our system is when we perform system-level tests before releasing a product. Thus software testing is a verification process for software quality assessment and improvement. Generally speaking, the activities for software quality assessment can be divided into two broad categories, namely, static analysis and dynamic analysis.

- **Static Analysis**: As the term “static” suggests, it is based on the examination of a number of documents, namely requirements

documents, software models, design documents, and source code. Traditional static analysis includes code review, inspection, walk-through, algorithm analysis, and proof of correctness. It does not involve actual execution of the code under development. Instead, it examines code and reasons over all possible behaviors that might arise during run time. Compiler optimizations are standard static analysis.

- Dynamic Analysis: Dynamic analysis of a software system involves actual program execution in order to expose possible program failures. The behavioral and performance properties of the program are also observed. Programs are executed with both typical and carefully chosen input values. Often, the input set of a program can be impractically large. However, for practical considerations, a finite subset of the input set can be selected. Therefore, in testing, we observe some representative program behaviors and reach a conclusion about the quality of the system. Careful selection of a finite test set is crucial to reaching a reliable conclusion.

2. Software Testing Techniques

2-1 Testing Objectives:

There are a number of rules that can serve as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.

2. A good test case is one that has a high probability of finding an as-yet undiscovered error.

3. A successful test is one that uncovers an as-yet-undiscovered error.

2-2 Testing Principles:

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Here is a suggested set of testing principles:

• All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the

most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete.
  Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- Testing should begin "in the small" and progress toward testing "in the large". The first tests planned and executed generally focus on individual components, as testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.
- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- To be most effective, testing should be conducted by an independent third party. By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing). The software engineer who created the system is not the best person to conduct all tests for the software.

Test Case Design:

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product (and most other things) can be tested in one of two ways:

1. Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function, This test is called ***black-box testing.***

2. Knowing the internal workings of a product, tests can be conducted to ensure that the internal operations are performed according to specifications and all internal components have been adequately exercised. This test is called ***white-box testing.***

When computer software is considered, black-box testing refers to tests that are conducted at the software interface. Although they are designed to uncover errors, black-box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information (e.g., a database) is maintained. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is established on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

At first glance it would seem that very thorough white-box testing would lead to "100 % correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. For example, consider the 100 line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10^{14} possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond.

So, working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. Exhaustive testing is impossible for large software systems.

White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be explored for validity. The attributes of both black- and white-box testing can be combined to provide an approach that validates the software interface and selectively ensures that the internal workings of the software are correct.

2-3 WHITE-BOX TESTING

White-box testing (sometimes called glass-box testing) is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that:

- Guarantee that all independent paths within a module have been exercised at least once,
- Workout all logical decisions on their true and false sides,
- Execute all loops at their boundaries and within their operational bounds, and
- Exercise internal data structures to ensure their validity.

A reasonable question might be posed here:

"Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?"

Stated another way, why don't we spend all of our energy on black-box tests?

The answer lies in the nature of software defects:

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement functions, conditions, or controls that are out of the mainstream. Everyday processing tends to be well understood, while "special case" processing tends to fall into the cracks.
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

Each of these reasons provides an argument for conducting white-box tests. Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White-box testing is far more likely to uncover them.

2-3-1 BASIS PATH TESTING

Basis path testing is a white-box testing technique. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

1- Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced.

The flow graph depicts logical control flow using the notation illustrated in Figure below. Each structured construct has a corresponding flow graph symbol.

To illustrate the use of a flow graph, we consider the procedural design representation in Figure A. Here, a flowchart is used to depict program control structure.

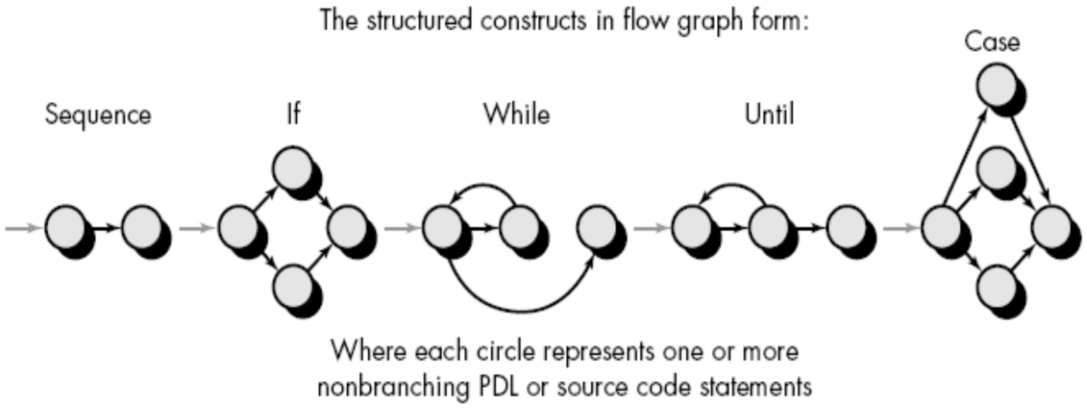


Figure B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

2- Cyclomatic Complexity

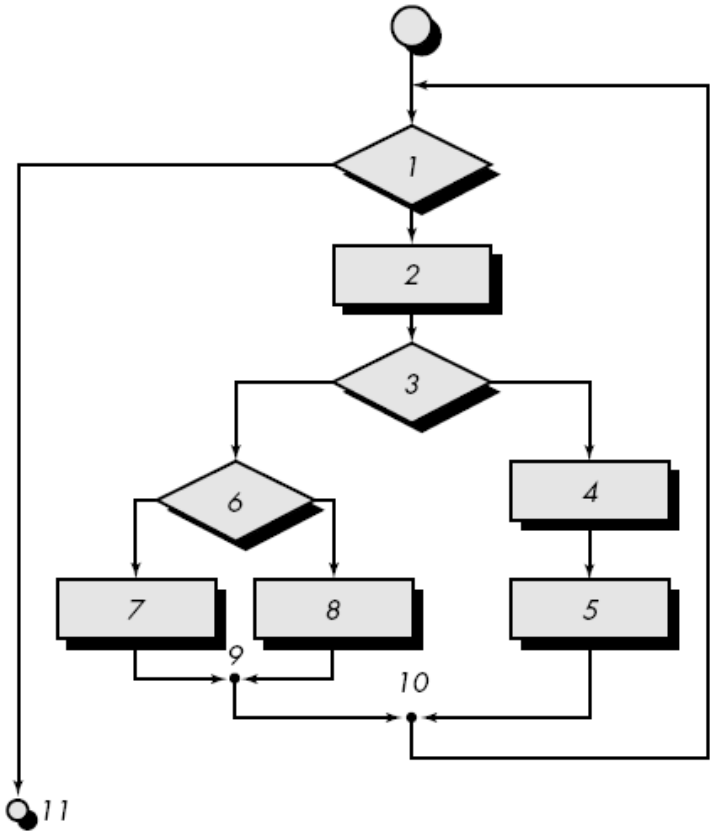
Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests

that must be conducted to ensure that all statements have been executed at least once.

An ***independent path*** is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in the Figure are:

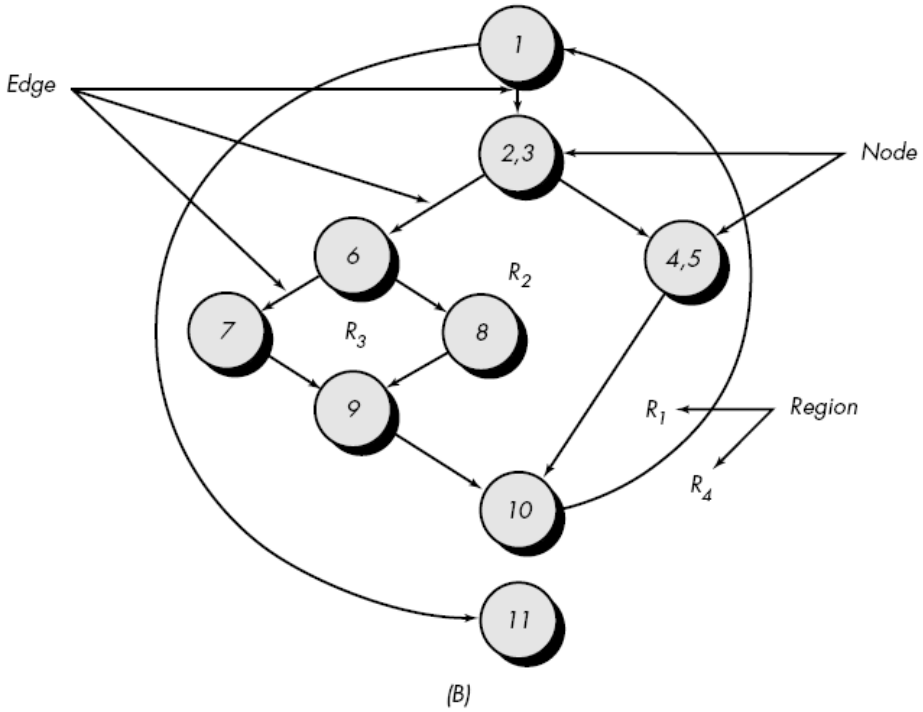
path 1: 1-11
path 2: 1-2-3-4-5-10-1-11
path 3: 1-2-3-6-8-9-10-1-11
path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge.



(A)

The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.



(B)

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity V(G) for a flow graph G is defined as:
   V(G) = (E – N) + 2
   where E is the number of flow graph edges,
   N is the number of flow graph nodes.
3. V(G) is also defined as **V(G) = P + 1**
   where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure B, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. V(G) = (Edges - Nodes) + 2 ➔ (11-9)+2=4.
3. V(G) = predicate nodes + 1 ➔ 3+1=4.

Therefore, the cyclomatic complexity of the flow graph in Figure B is 4.

Each node that contains a condition is called a **predicate node** and is characterized by two or more edges emanating from it.

More important, the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set, and thus an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

3- Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to derive the basis set:

1. **Using the design or code as a foundation, draw a corresponding flow graph.** Referring to the PDL (Program Design Language) for average in Figure below, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes.

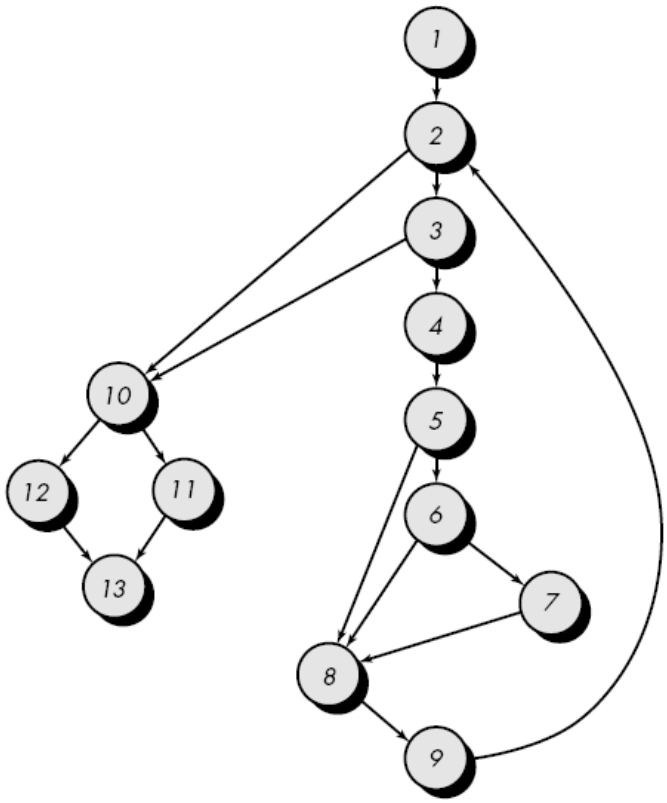
```
PROCEDURE average;

  * This procedure computes the average of 100 or fewer
    numbers that lie between bounding values; it also computes the
    sum and the total number valid.

INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid;
   minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
1  i = 1;
   total.input = total.valid = 0;        2
   sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100   3
4     increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5        THEN increment total.valid by 1;
7             sum = s sum + value[i]
         ELSE skip
      ENDIF
8     increment i by 1;
9  ENDDO
   IF total.valid > 0   10
11    THEN average = sum / total.valid;
12    ELSE average = -999;
13 ENDIF
END average
```

2**. Determine the cyclomatic complexity** of the resultant flow graph:
V(G) = 6 regions
V(G) = 17 edges _ 13 nodes + 2 = 6
V(G) = 5 predicate nodes + 1 = 6



3. **Determine a basis set of linearly independent paths.** The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six paths:

path 1: 1-2-10-11-13
path 2: 1-2-10-12-13
path 3: 1-2-3-10-11-13
path 4: 1-2-3-4-5-8-9-2-. . .
path 5: 1-2-3-4-5-6-8-9-2-. . .
path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

Path 1 test case:
value(k) = valid input, where k < i for $2 \leq i \leq 100$
value(i) = -999 where $2 \leq i \leq 100$
Expected results: Correct average based on k values and proper totals.
Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:
value(1) = -999
Expected results: Average = -999; other totals at initial values.

Path 3 test case:
Attempt to process 101 or more values.
First 100 values should be valid.
Expected results: Same as test case 1.

Path 4 test case:
value(i) = valid input where i < 100
value(k) < minimum where k < i
Expected results: Correct average based on k values and proper totals.

Path 5 test case:
value(i) = valid input where i < 100
value(k) > maximum where k <= i
Expected results: Correct average based on n values and proper totals.

Path 6 test case:
value(i) = valid input where i < 100
Expected results: Correct average based on n values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

2-3-2 Control Structure Testing:

1- Condition Testing:

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬).

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies generally have two advantages:

1- Measurement of test coverage of a condition is simple.
2- The test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program P is effective for detecting errors in the conditions contained in P, it is likely that this test set is also effective for detecting other errors in P.

Domain Testing requires three or four tests to be derived for a relational expression.

For a relational expression of the form: **E1 <relational-operator> E2**, three tests are required to make the value of E1 greater than, equal to, or less than that of E2. This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if n is small.

2- Data Flow Testing:

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. Assume that each statement in a program is assigned a distinctive statement number and that each function does not modify its parameters or global variables. For a statement with S as its statement number,

DEF(S) = {X | statement S contains a definition of X}

USE(S) = {X | statement S contains a use of X}

If statement S is an (if) or (loop) statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

One simple data flow testing strategy requires every DU chain be covered at least once. DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the (then) part has no definition of any variable and the (else) part does not exist. In this situation, the else branch of the (if) statement is not necessarily covered by DU testing.

Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

3- Loop Testing:

Loops are the foundation 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.**

3-1 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.

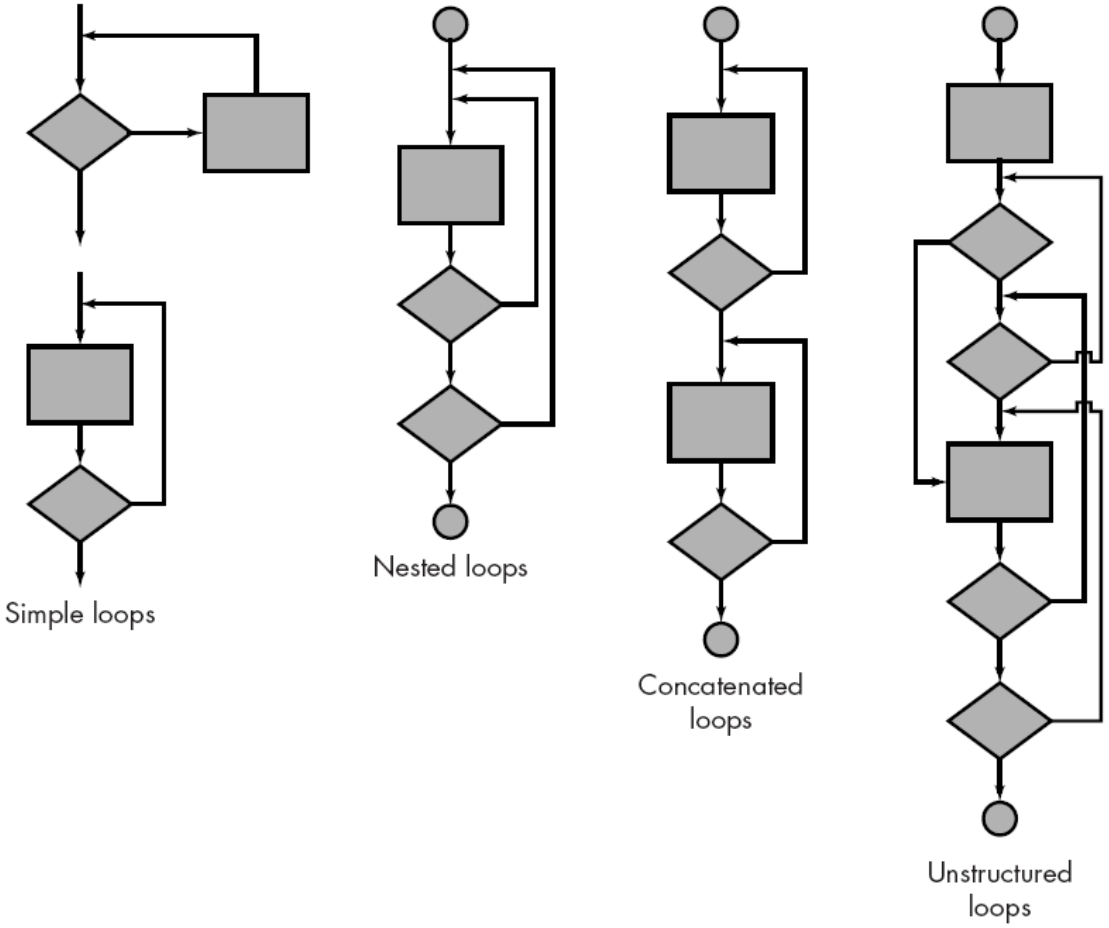
3-2 Nested loops.

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 (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.

3-3 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.

3-4 Unstructured loops.

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

3. Software Quality Assurance Activities

Software Quality Assurance makes sure that the software is free from defects or mistakes and performs all the functionalities without complaints just before the delivery.

The Software Quality Assurance is measured based on the internal and external quality features of the software. The external quality is measured based on the real-time activities in operational mode and how the software is useful for the end users.

The internal quality is measured based on the style and quality of the code written. Mostly the client will bother about the external quality only. But, in effect for a perfect performance of the software, the internal quality is an important aspect to be considered and maintained.

The 2 approaches to determine the Software Quality Assurance are:

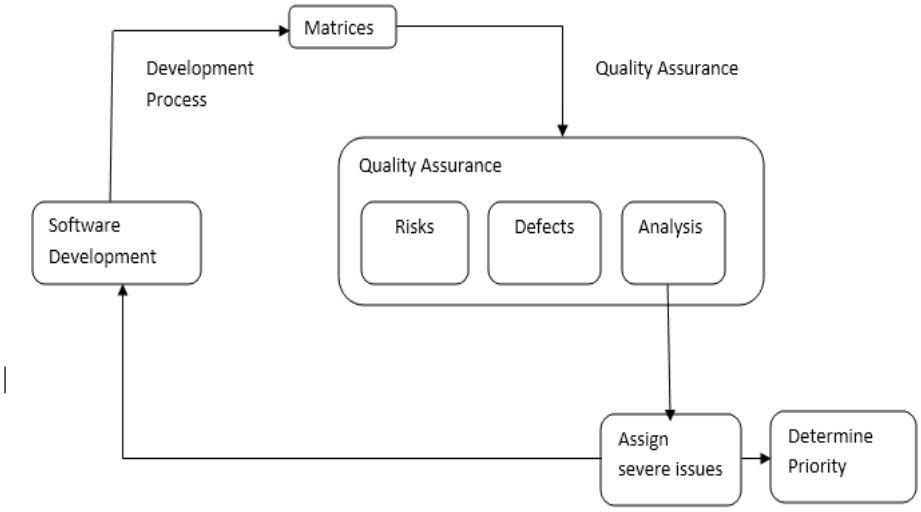
1-The Defect Management Approach

The defects are categorized on the basis of the severity. The counts of the defects are taken and the actions are decided by analyzing the occurrence of defects. The defects come from very minute issues and extend to the coding defects, the non-completion of the requirements and of course if the application does not look good for the customers.

Defect management process is based on some principles:

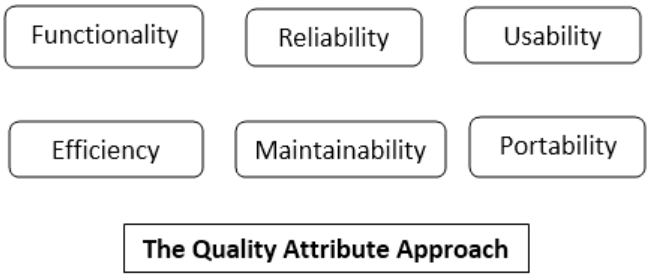
- Preventing defect is the primary goal of defect management approach. But preventing defects completely is not possible and so the purpose is to find out the defects as early as possible and to minimize their impact.
- To prevent the defects some process should be altered.
- The defect measurement processes should be integrated into the software development process, and thereby the process can be improved.
- Defect information always helps to improve the processes and hence it is very useful for perfect completion of the software developed.

The diagram below explains the various stages of the defect management approach.



2-The Software Quality Assurance Attribute Approach

There is a list of attributes which describes the step by step approach to obtain SQA. The attributes are given as in the diagram below:



SQA Activities to Assure the Software Quality:

The SQA of the software is analyzed and ensured by performing a series of activities. The activities are performed as step by step process and the result analysis is reported for the final evaluation process.

1- A Quality Management Plan

It is designed and developed for the SQA Process. The plan includes the proper technical methods to manage the SQA activities.

2- Applying Software Engineering Techniques

The software engineering techniques are selected to achieve software quality. The techniques to be used for SQA are determined by analyzing the requirements collected. Also, a project estimate is prepared.

3- Technical Reviews

The Formal Technical Reviews (FTRs) are conducted to assess the quality and design of the Quality Management Plan.

4- Applying the Testing Strategy

The testing strategy is designed and applied. The various levels of testing are designed and scheduled. The testing strategies are designed based on the policies of the company; the stages for each test phase execution are designed and scheduled for the concerned persons.

5- Ensuring Process Adherence

The process adherence is the combination of two tasks: product evaluation and process monitoring. **Product evaluation** is the process of ensuring all the requirements identified in the product development result to the completion of the functionalities. **Process Monitoring** is the process of comparing actual steps for the procedures with the expected steps designed in the documented procedures.

6- The Change Control Process

The Change Control is the process which formalizes the request for changes, evaluates the quality/nature of changes, and controls the impact of changes. The Change Control Mechanism is designed and implemented during the design and development stages.

7- Software Quality Assurance Audits

SQA Audits inspects the Software Development Process by comparing to the established processes. SQA Auditor is the responsible person who reviews and checks the activities are executed to the highest possible standards. The quality of the project handling can be analyzed only through the results of the review submitted by the SQA Auditor.

8- Generate Reports

Appropriate records for all the activities should be generated for future references. These activities evaluate the quality of a project and also tests the way of handling project management processes. This will result in a review of the performance of the Test Engineer who is in charge of the Test Management Processes.

4. Software Testing Strategies

Testing is a set of activities that can be planned in advance and conducted systematically. A number of software testing strategies have been proposed in the literature. For this reason a prototype for software testing should be defined for the software process.

All the software testing strategies have the following generic characteristics:

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

4.1. Verification and Validation

Software testing is one element of a more general 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).

4.2. A Software Testing Strategy

The software engineering process may be viewed as the spiral shape illustrated in the 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.

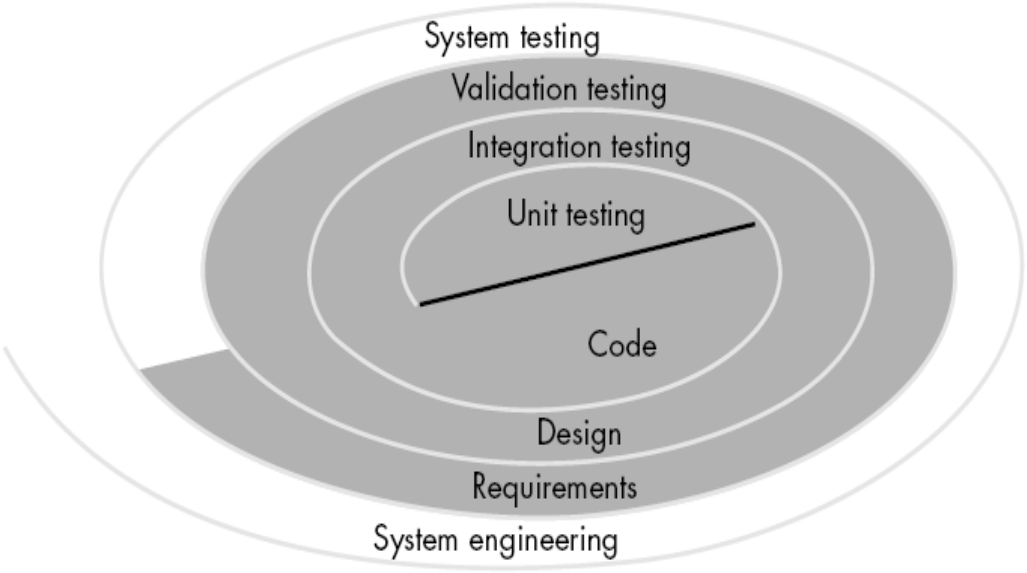


Fig. Testing Strategy

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.

4.3.1 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 unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure below. 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 error handling paths are tested.

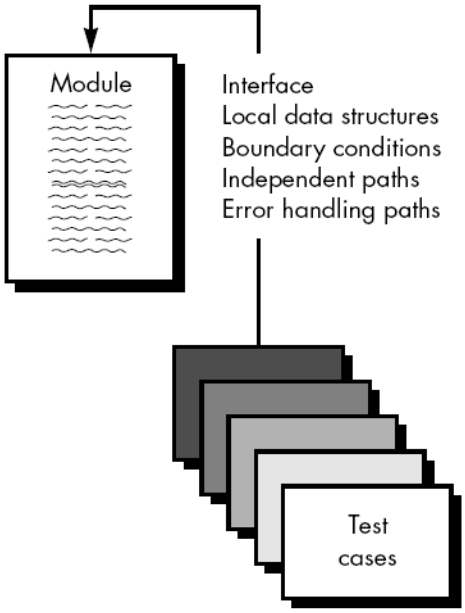


Fig. Unit Testing

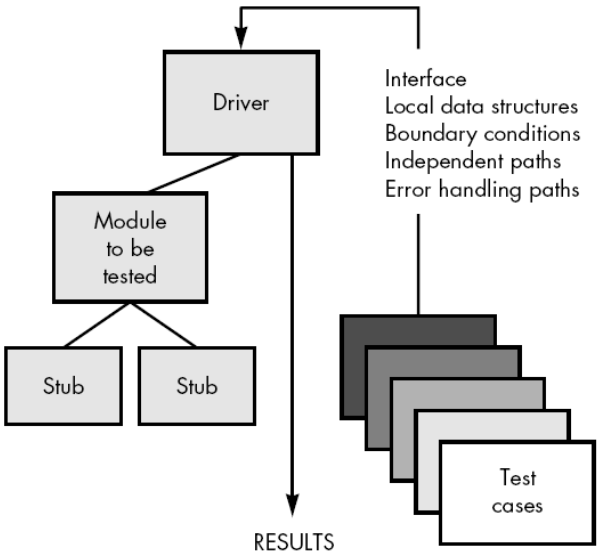
Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Unit Test Procedures

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins.

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure below. In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product.



Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

4.3.2 INTEGRATION TESTING

One might ask a seemingly legitimate question: once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?"

The problem, of course, is "putting them together" —interfacing. Data can be lost across an interface; one module can have an unintentional, adverse effect on another; sub-functions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the 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 is often a tendency to attempt *non-incremental* integration; that is, all components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; in the sections that follow, a number of different incremental integration strategies are discussed.

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 below, *depth-first integration* would integrate all components on a major control path of the structure. For example, selecting the left hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and 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 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

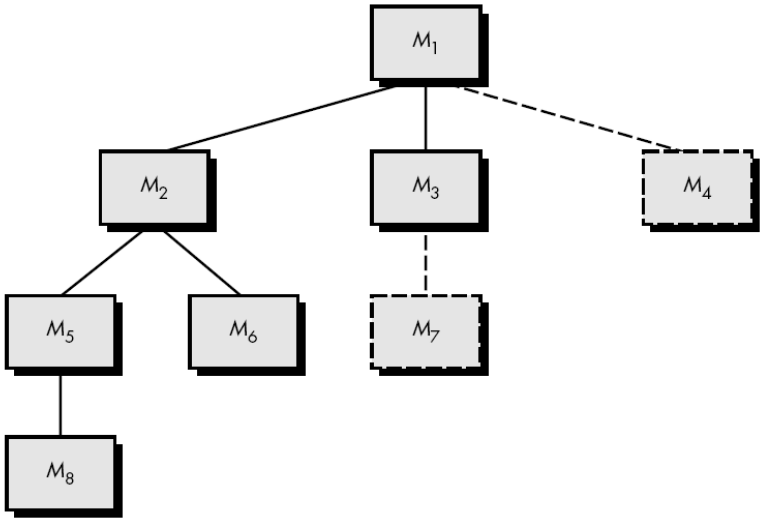


Fig. Top-Down Integration

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

Bottom-up Integration

This strategy begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub-function.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in the figure below. 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 M_a. Drivers D1 and D2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

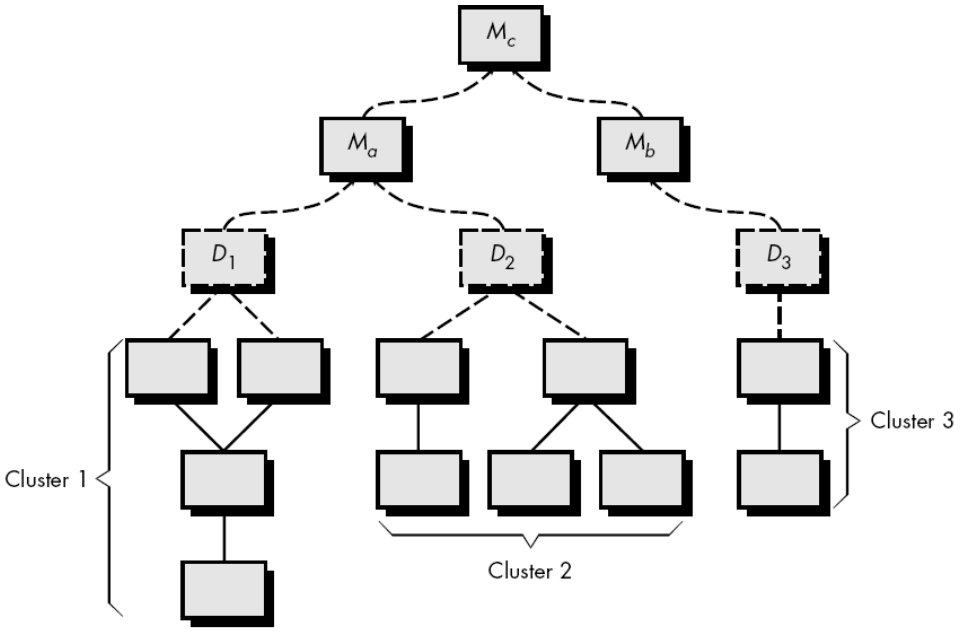


Fig. Bottom-up Integration

Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects*.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the nu mber of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Comments on Integration Testing

There has been much discussion of the relative advantages and disadvantages of top-down versus bottom-up integration testing. In general, the advantages of one strategy tend to result in disadvantages for the other strategy.

The major disadvantage of the **top-down approach** is the need for stubs and the attendant testing difficulties that can be associated with them.

The major disadvantage of **bottom-up** integration is that "the program as an entity does not exist until the last module is added". This drawback is tempered by easier test case design and a lack of stubs.

Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called *sandwich testing*) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

As integration testing is conducted, the tester should identify *critical modules.* A critical module has one or more of the following characteristics:

1) addresses several software requirements,
2) has a high level of control (resides relatively high in the program structure),

3) is complex or error prone, or
4) has definite performance requirements.

Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

4.3.3 VALIDATION TESTING

At the conclusion of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests—*validation testing*—may begin. Validation can be defined in many ways, but a simple definition is that validati*on succeeds when software functions in a manner that can be reasonably expected by the customer*.

Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that:

- all functional requirements are satisfied,
- all behavioral characteristics are achieved,
- all performance requirements are attained,
- documentation is correct, and
- human engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exist:

(1) The function or performance characteristics conform to specification and are accepted, or
(2) a deviation from specification is uncovered and a *deficiency list* is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be incomprehensible to a user in the field.

When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements.

Most software product builders use a process called **alpha and beta testing** to uncover errors that only the end-user seems able to find.

The ***alpha test*** is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The ***beta test*** is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

4.3.4 SYSTEM TESTING

Software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem.

The software engineer should anticipate potential interfacing problems and

(1) design error-handling paths that test all information coming from other elements of the system,
(2) conduct a series of tests that simulate bad data or other potential errors at the software interface,
(3) record the results of tests to use as "evidence" if finger-pointing does occur, and
(4) participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

Recovery Testing

Many computer based systems must recover from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

During security testing, the tester plays the role of the individual who desires to penetrate the system. The tester may attempt to acquire passwords, may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery;

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations.

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example,

1- special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
2- input data rates may be increased by an order of magnitude to determine how input functions will respond,
3- test cases that require maximum memory or other resources are executed,
4- test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. *Performance testing* is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

5. SOFTWARE REVIEWS

Software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities that we have called *analysis, design,* and *coding.*

Many different types of reviews can be conducted as part of software engineering. Each has its place. An informal meeting around the coffee machine is a form of review, if technical problems are discussed. A formal presentation of software design to an audience of customers, management, and technical staff is also a form of review.

- **Formal Technical Reviews (FTR)**

A formal technical review is a software quality assurance activity performed by software engineers (and others). The objectives of the FTR are to:

(1) uncover errors in function, logic, or implementation for any representation of the software;
(2) verify that the software under review meets its requirements;
(3) ensure that the software has been represented according to predefined standards;
(4) achieve software that is developed in a uniform manner; and
(5) make projects more manageable.

- **Informal Technical Reviews (ITR)**

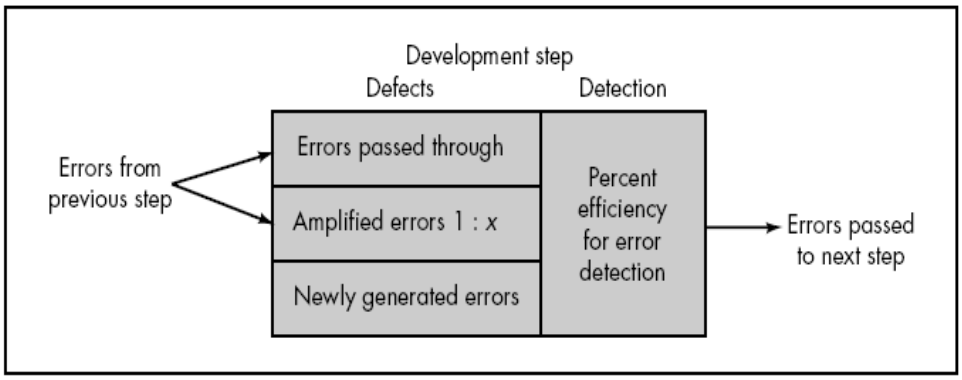
ITRs allow more casual approach in which one or more reviewers help the software engineer improve his work products. Typically, these meeting do not follow any particular plan and no formal minutes are recorded, pair programming is example on the review oriented aspects.

5. Defect Amplification and Removal:

A defect amplification model can be used to illustrate the **generation and detection of errors** during the preliminary design, detail design, and coding steps, of the software engineering process. The model is illustrated

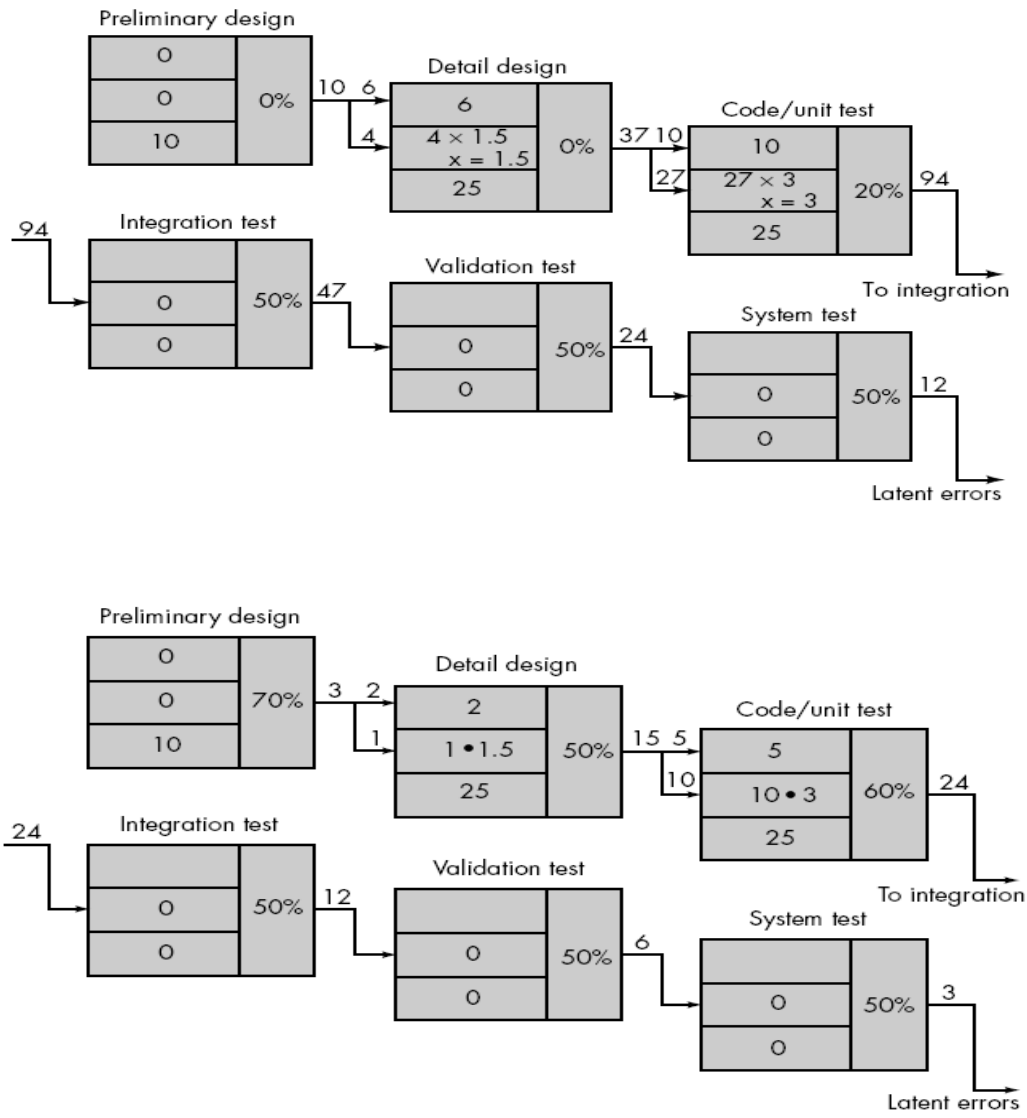
schematically in the figure below. A box represents a software development step. During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through.

In some cases, errors passed through from previous steps are amplified (amplification factor, *x*) by current work. The box subdivisions represent each of these characteristics and the percent of efficiency for detecting errors, a function of the thoroughness of the review.



First figure below illustrates a hypothetical example of defect amplification for a software development process in which no reviews are conducted. Referring to the figure, each test step is assumed to uncover and correct 50 percent of all incoming errors without introducing any new errors. Ten preliminary design defects are amplified to 94 errors before testing commences. Twelve latent errors are released to the field. Second Figure considers the same conditions except that design and code reviews are conducted as part of each development step. In this case, ten initial preliminary design errors are amplified to 24 errors before testing commences and only three latent errors exist.

To conduct reviews, you must expend time and effort, and your development organization must spend money. However, the results of the preceding example leave little doubt that you can pay now, or pay much more later.



REVIEW METRICS

Technical reviews are one of many actions that are required as part of good software engineering practice. Each action requires dedicated human effort.

Before analysis can begin, a few simple computations must occur. The total review effort and the total number of errors discovered are defined as:

- $$E_{review} = E_p + E_a + E_r$$

Where Preparation effort, Ep — the effort (in person-hours) required to review a work product prior to the actual review meeting, and Assessment

effort, Ea— the effort (in person-hours) that is expended during the actual review, and Rework effort, Er — the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review.

- $$\text{Err}_{\text{tot}} = \text{Err}_{\text{minor}} + \text{Err}_{\text{major}}$$

Where Err minor represent Minor errors found,—the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct) , and Err major represent Major errors found,—the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct).

Error density represents the errors found per unit of work product reviewed.

- $$\text{Error density} = \frac{\text{Err}_{\text{tot}}}{\text{WPS}}$$

Where Work product size, WPS—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)

For example, if a requirements model is reviewed to uncover errors, inconsistencies, and omissions, it would be possible to compute the error density in a number of different ways. The requirements model contains 18 UML diagrams as part of 32 overall pages of descriptive materials. The review uncovers 18 minor errors and 4 major errors.

Therefore, $\text{Err}_{\text{tot}} = 22$. Error density is 1.2 errors per UML diagram or 0.68 errors per requirements model page.

If reviews are conducted for a number of different types of work products (e.g., requirements model, design model, code, test cases), the percentage of errors uncovered for each review can be computed against the total number of errors found for all reviews. In addition, the error density for each work product can be computed.

Once data are collected for many reviews conducted across many projects, average values for error density enable you to estimate the number of errors to be found in a new (as yet unreviewed document). For example, if the average error density for a requirements model is 0.6 errors per page, and a new requirement model is 32 pages long, a rough estimate suggests that

your software team will find about 19 or 20 errors during the review of the document. If you find only 6 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough.

Once testing has been conducted, it is possible to collect additional error data, including the effort required to find and correct errors uncovered during testing and the error density of the software. The costs associated with finding and correcting an error during testing can be compared to those for reviews.

6. The SQA Plan:

The SQA Plan provides a road map for instituting software quality assurance. The plan serves as a template for SQA activities that are instituted for each software project.

A standard for SQA plans has been published by the IEEE. The standard recommends a structure that identifies:

(1) the purpose and scope of the plan,
(2) a description of all software engineering work products (e.g., models, documents, source code) that fall within the purview of SQA,
(3) all applicable standards and practices that are applied during the software process,
(4) SQA actions and tasks (including reviews and audits) and their placement throughout the software process,
(5) the tools and methods that support SQA actions and tasks,
(6) software configuration management procedures,
(7) methods for assembling, safeguarding, and maintaining all SQA-related records, and
(8) Organizational roles and responsibilities relative to product quality.