SOFTWARE 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.

These objectives imply a dramatic change in viewpoint. They move counter to the commonly held view that a successful test is one in which no errors are found. Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

If testing is conducted successfully, it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met. In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

SOFTWARE TESTING : PRINCIPLES

(1) All tests should be traceable to customer requirements. The most severe defects are those that cause the program to fail to meet its requirements.

(2) 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.

(3) The Pareto principle applies to software testing. The Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

(4) 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.

(5) Exhaustive testing is not possible.

(6) To be most effective testing should be conducted by an independent third party.

SOFTWARE TESTABILITY

Software testability is simply how easily a computer program can be tested.

Characteristics that lead to testable software

(1) Operability. "The better it works, the more efficiently it can be tested."

(2) Observability. "What you see is what you test."

(3) Controllability. "The better we can control the software, the more the testing can be automated and optimized.

(4) Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

(5) Simplicity. "The less there is to test, the more quickly we can test it."

(6) Stability. "The fewer the changes, the fewer the disruptions to testing."

(7) Understandability. "The more information we have, the smarter we will test."

TEST CASE DESIGN

(1) BLACK-BOX TESTING: 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.

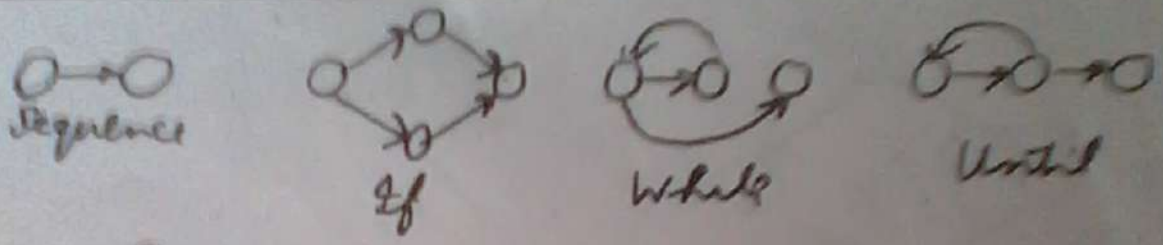
(2) WHITE-BOX TESTING: Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

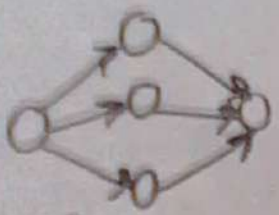
It is not possible to exhaustively test every program path because the number of paths is simply too large.

BASIS PATH TESTING

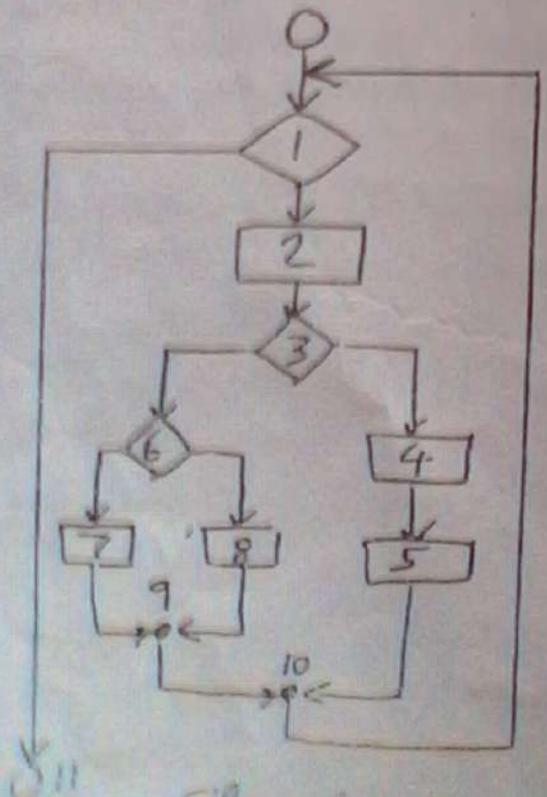
It is a white-box testing technique.
It enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

FLOW GRAPH NOTATION

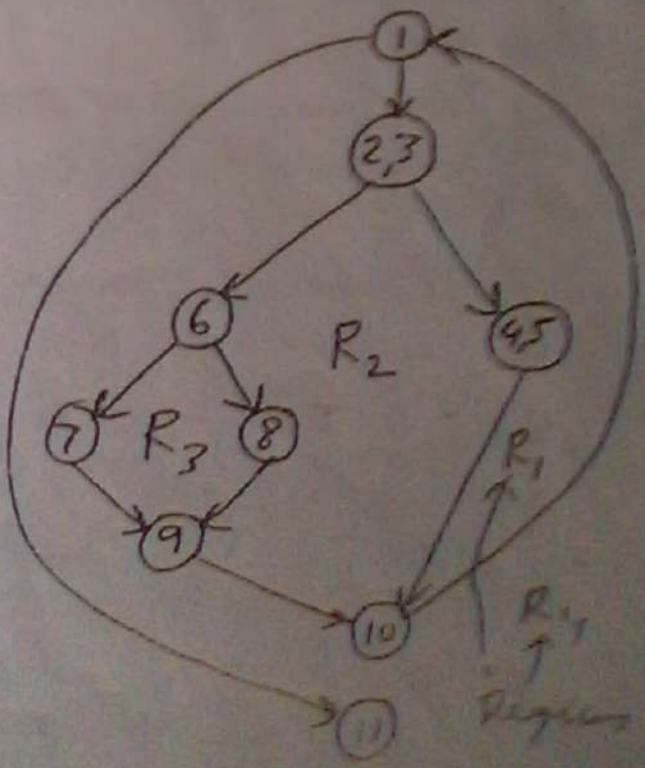




Each circle represents one or more nonbranching source code statements.

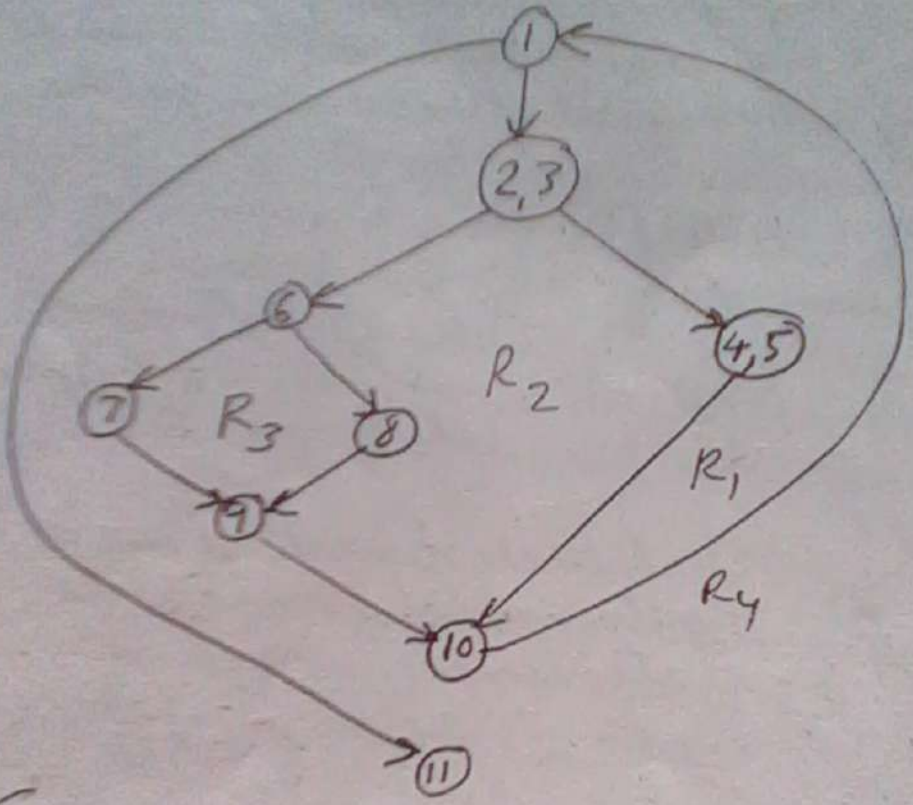


Flowchart



Flow graph

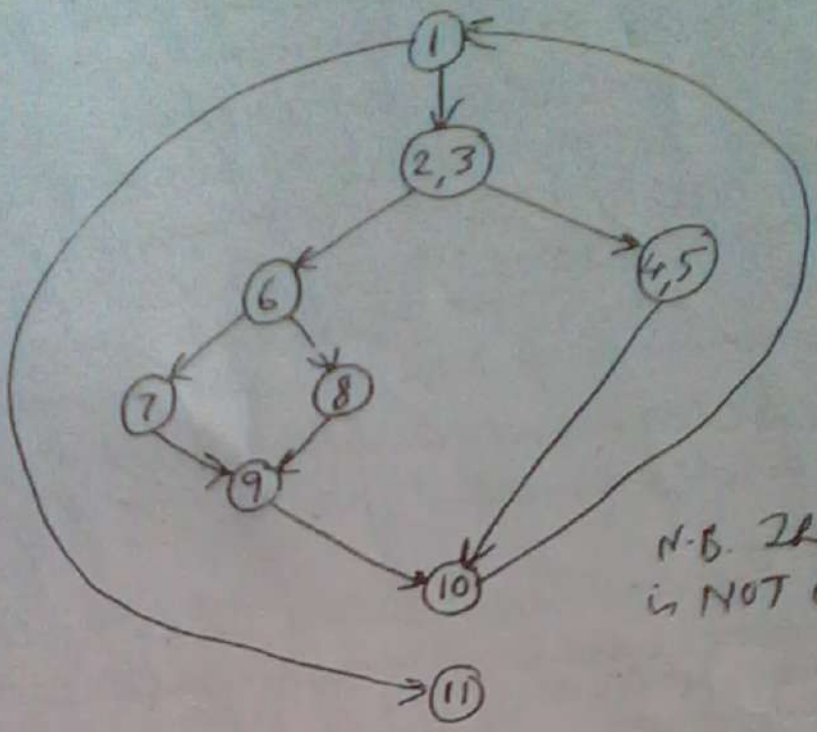
Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.



Cyclomatic complexity is measured in one of the ways:

1. The number of regions of the flow graph correspond to the cyclomatic complexity. 4
2. Cyclomatic complexity, $V(G)$ for a flow graph G, is defined as
   $V(G) = E - N + 2$ $11 - 9 + 2 = 4$
3. $V(G) = P + 1$
   where P is the number of predicate nodes contained in the flow graph G. $3 + 1 = 4$

BASIS SET



N.B. The basis set is NOT unique.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
Flow graph → an independent path must move along at least one edge that has not been traversed before the path is defined.

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

} A set of independent paths.

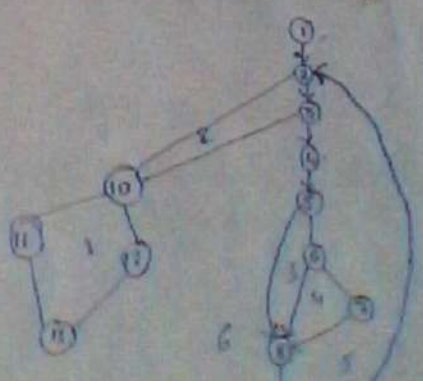
Paths 1, 2, 3, and 4 constitute a basis set for the flow graph. That is, if tests can be designed to force execution of these paths, every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.

DERIVING TEST CASES

1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph. This value, V(G), provides the number of linearly independent paths through the program control structure.
3. Determine a basis set of linearly independent paths.
4. Prepare test cases that will force execution of each path in the basis set.

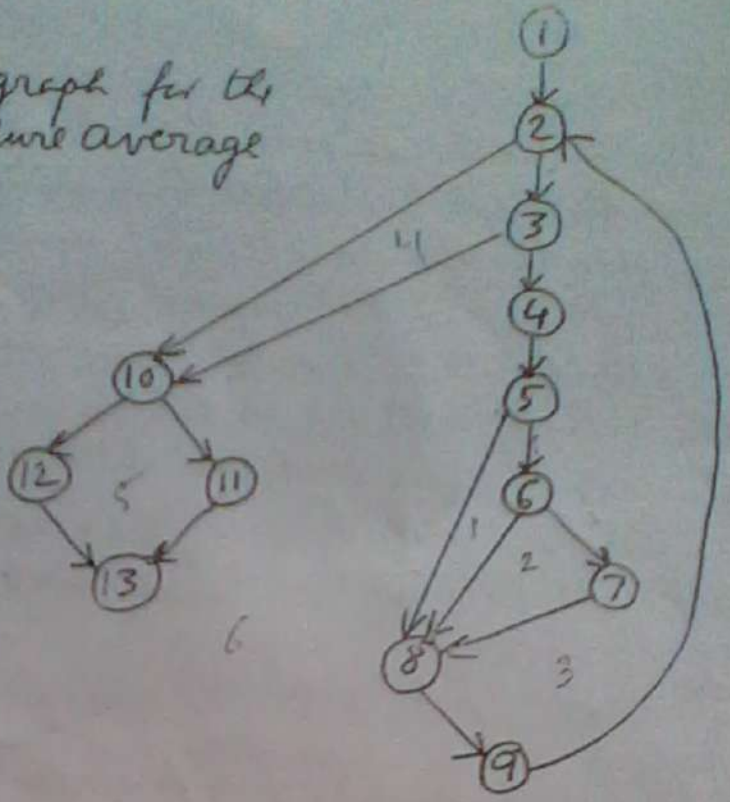
DERIVING TEST CASES: Example Source Code

```
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;
    sum = 0;
    DO WHILE (value[i] <> -999) AND (total.input < 100)   (2) (3)
(4)   increment total.input by 1;
      IF (value[i] >= minimum) AND (value[i] <= maximum)   (5) (6)
(7)     THEN increment total.valid by 1;
             sum = sum + value[i]
        ELSE skip
(8)   ENDIF
      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
```



DERIVING TEST CASES (contd.)

Flow graph for the procedure average



Cyclomatic complexity

V(G) = 6 regions
V(G) = 17 edges − 13 nodes + 2 = 6
V(G) = 5 predicate nodes + 1 = 6

A basis set of linearly independent 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-...

... indicates that any path through the remainder of the control structure is acceptable.

PREPARING TEST CASES THAT FORCE EXECUTION OF EACH PATH in the BASIS SET

Path 1 test case:

End of list encountered.
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.

Path 2 test case:

End of list encountered AND no input is valid, i.e. no input is in the range [minimum, maximum]

value (1) = -999
Expected results: Average = -999.
Other totals at initial values.

Path 3 test case:

Total number of inputs is 101 or more.
First 100 values should be valid.
Expected results: Same as test case 1.

Path 4 test cases

A value is less than the lower bound.
value (i): Next data to be processed.
value (k) < minimum where $k \leq i$
Expected results: Correct average based on k values and proper totals

Path 5 test cases:

A value is greater than the upper bound.
value (i) = valid input where $i \leq 100$
value (k) > maximum where $k \leq i$.
expected results: Correct average based on k values and proper totals

Path 6 test cases:

Valid input.
value (k) = valid input where $k < 100$.
Expected results: Correct average based on k 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.

TEST STUBS AND DRIVERS

Executing test cases on single components or combinations of components requires the tested component to be isolated from the rest of the system. Test drivers and test stubs are used to substitute for missing parts of the system.

A test driver simulates the part of the system that calls the component under test. It passes the test inputs identified in the test case analysis to the component and displays the results

A test stub simulates components that are called by the tested component. The test stub must provide the same API as the method of the simulated component and must return a value compliant with the return result type.

UNIT TESTING

Unit testing focuses on the building blocks of the software system, that is, objects and subsystems.

- It reduces the complexity of overall test activities, allowing us to focus on smaller units of the system.
- Unit testing makes it easier to pinpoint and correct faults given that few components are involved in the test.
- It allows parallelism in the testing activities: that is, each component can be tested independently of the others.

Techniques (the important ones)

1. Equivalence testing
2. Boundary testing
3. Path testing
4. State-based testing

EQUIVALENCE TESTING

This blackbox testing technique minimizes the number of test cases. The possible inputs are partitioned into equivalence classes, and a test case is selected for each class.
The assumption of equivalence testing is that systems usually behave in similar ways for all members of a class. To test the behaviour associated with an equivalence class, we only need to test one member of the class.

Equivalence testing consists of two steps: identification of the equivalence classes and selection of the test inputs.

Criteria used in determining equivalence classes

- Coverage – Every possible input belongs to one of the equivalence classes.
- Disjointedness – No input belongs to more than one equivalence class.
- Representation – If the execution demonstrates an erroneous state when a particular member of an equivalence class is used as input then the same erroneous state can be detected by using any other member of the class as input.

EQUIVALENCE TESTING (cont'd.)

```
class MyGregorianCalendar {
...
    public static int getNumDaysInMonth
                        (int month, int year) {...}
}
```

Interface for a method computing the number of days in a given month.

We find three equivalence classes for the month parameter:

- months with 31 days (i.e., 1, 3, 5, 7, 8, 10, 12)
- months with 30 days (i.e. 4, 6, 9, 11)
- February, which can have 28 or 29 days.

Similarly we find two equivalence classes for the year: leap years and non-leap years.

Equivalence class	Month	Year
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

Equivalence classes and selected valid inputs for testing the getNumDaysInMonth() method.

BOUNDARY TESTING

This special case of equivalence testing focused on the conditions at the boundary of the equivalence classes. Rather than selecting any element in the equivalence class, boundary testing requires that the elements be selected from the "edges" of the equivalence class. The assumption behind boundary testing is that developers often overlook special cases at the boundary of the equivalence classes (e.g., 0, empty strings, year 2000).

Equivalence class	Month	Year
Leap year divisible by 400	2 (February)	2000
Non-leap year divisible by 100	2 (February)	1900
Nonpositive invalid months	0	1291
Positive invalid months	13	1315

Additional boundary cases selected for the getNumDaysInMonth() method.

In general, years that are multiples of 4 are leap years. Years that are multiples of 100, however, are not leap years, unless they are also multiples of 400. For example, 2000 was a leap year, whereas 1900 was not.

A disadvantage of equivalence class and boundary testing is that these techniques do not explore combinations of test input data. In many cases, a program fails because a combination of certain values causes the fault.

Unit testing focuses on individual components. The developer discovers faults using equivalence testing, boundary testing, path testing, and other methods. Once faults in each component have been removed and the test cases do not reveal any new fault, components are ready to be integrated into larger subsystems. At this point, components are still likely to contain faults, as test stubs and drivers used during unit testing are only approximations of the components they simulate. Moreover, unit testing does not reveal faults associated with the component interfaces resulting from invalid assumptions when calling these interfaces.

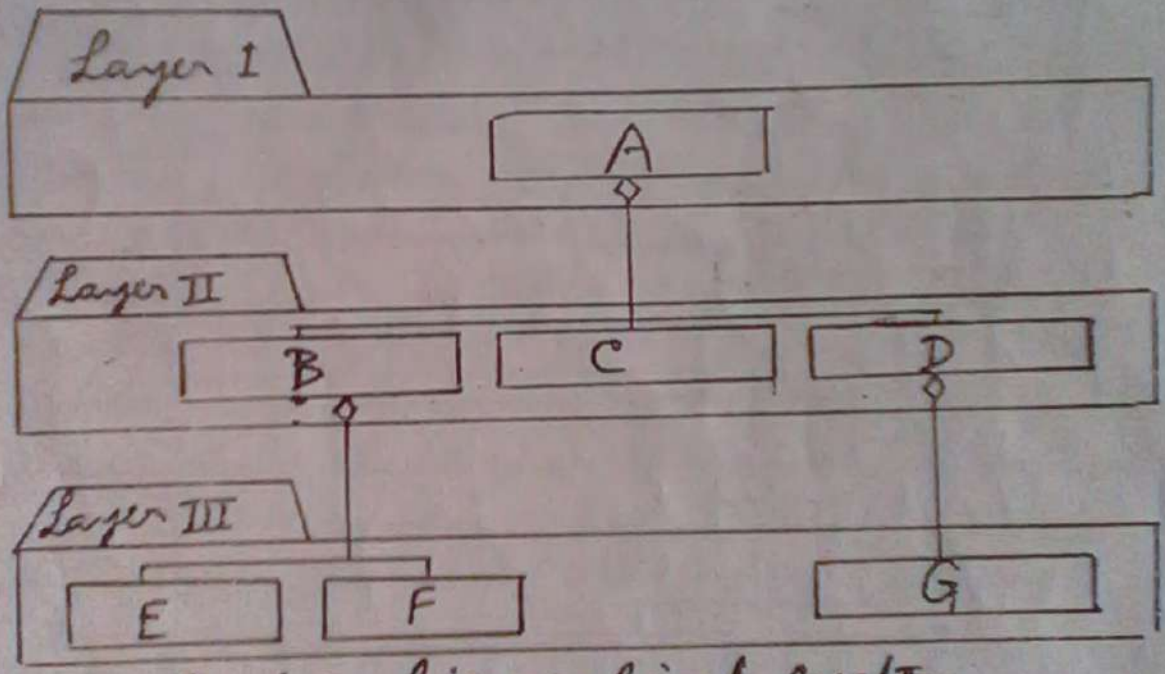
(Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Two or more components are integrated and tested and when no new faults are revealed additional components are added to the group. This procedure allows the testing of increasingly more complex parts of the system while keeping the location of potential faults relatively small (i.e., the most recently added component is usually the one that triggers the most recently discovered faults).

INTEGRATION TESTING STRATEGIES

Several approaches have been devised to implement an integration testing strategy:

- big bang testing
- bottom-up testing
- top-down testing
- sandwich testing.

Each of these strategies was originally devised by assuming that the system decomposition is hierarchical.

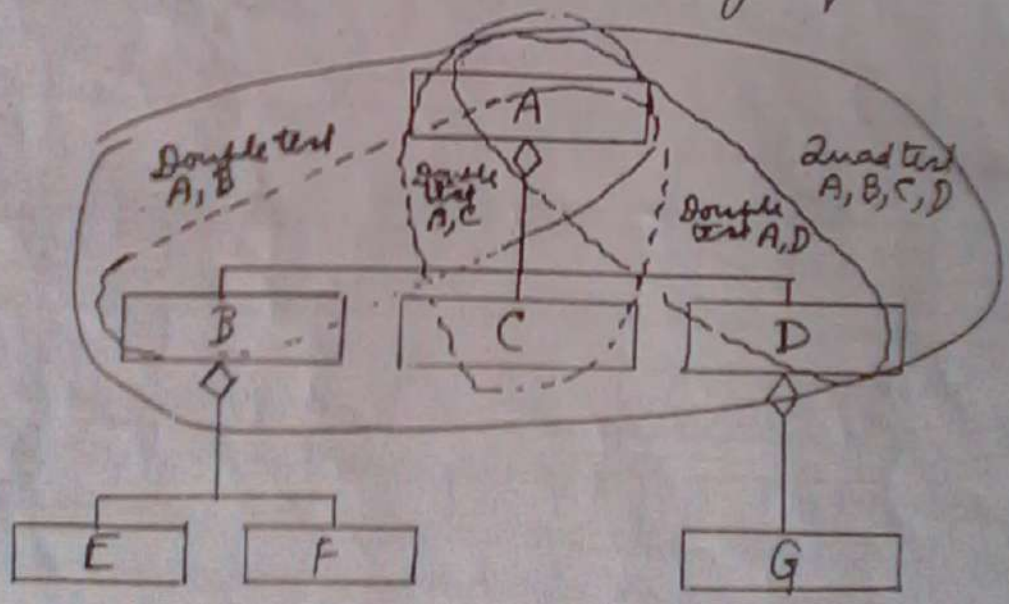


Example of a hierarchical system decomposition using three layers.

The big bang testing strategy assumes that all components are first tested individually and then tested together as a single system. The advantage is that no additional test stubs or drivers are needed. Although this strategy sounds simple, big bang testing requires

Integration testing strategies (contd.)

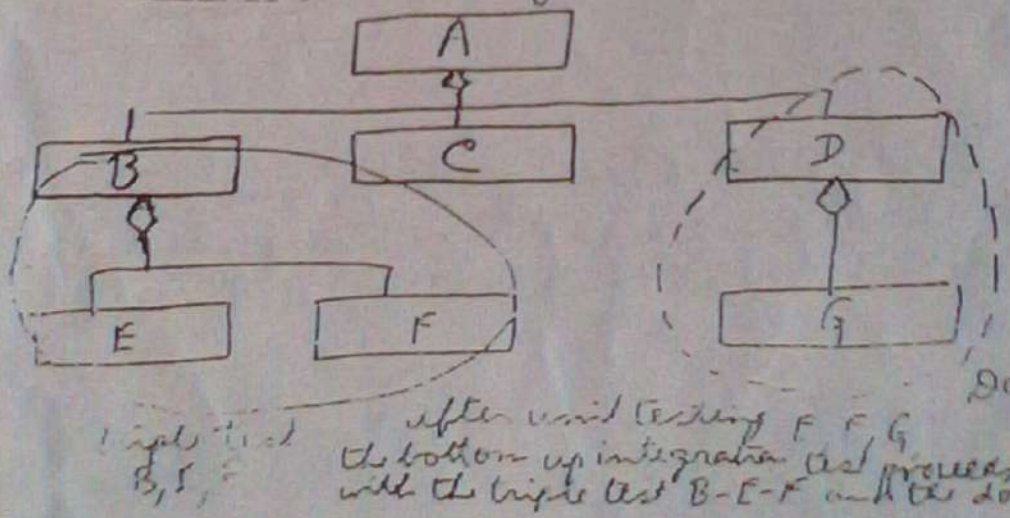
The top-down testing strategy unit tests the components of the top layer first, and then integrates the components of the next layer down. When all components of this new layer have been tested together the next layer is selected. Again, the tests incrementally add one component at a time. This is repeated until all layers are combined and involved in the test. Test stubs are used to simulate the components of lower layers that have not yet been integrated. Test drivers are not needed during top-down testing.



After unit testing subsystem A, the integration test proceeds with the double tests A-B A-C and A-D, followed by the quad test A-B-C-D

BIG BANG: If a test uncovers a failure it is impossible to distinguish failures in the interface from failures within a component. Moreover, it is difficult to pinpoint the specific component (or combination of components) responsible for the failure, as all components in the system are exercised.

The bottom-up testing strategy first tests each component of the bottom layer individually and then integrates them with components of the next layer up. If two components are tested together, we call this a double test. Testing three components together is a triple test, and a test with four components is called a quadruple test. This is repeated until all components from all layers are combined. Test drivers are used to simulate the components of higher layers that have not yet been integrated. No test stubs are necessary during bottom-up testing.



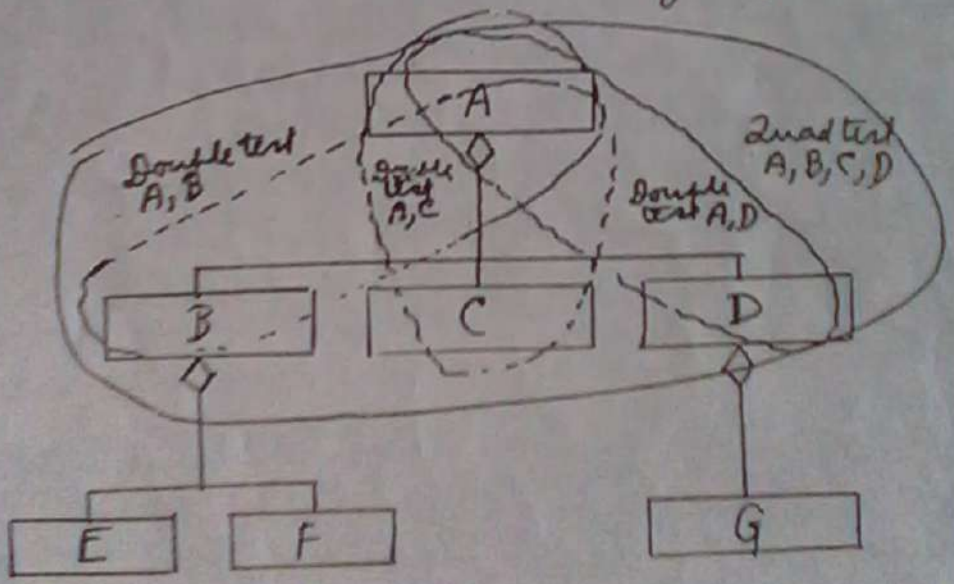
Double test D, G

Triple test B, E, F

after unit testing F, G the bottom-up integration test proceeds with the triple test B-E-F and the double test D-G

Integration testing strategies (cont'd.)

The top-down testing strategy unit tests the components of the top layer first, and then integrates the components of the next layer down. When all components of this new layer have been tested together, the next layer is selected. Again, the tests incrementally add one component at a time. This is repeated until all layers are combined and involved in the test. Test stubs are used to simulate the components of lower layers that have not yet been integrated. Test drivers are not needed during top-down testing.



After unit testing subsystem A, the integration test proceeds with the double tests A-B, A-C, and A-D, followed by the quad test A-B-C-D.

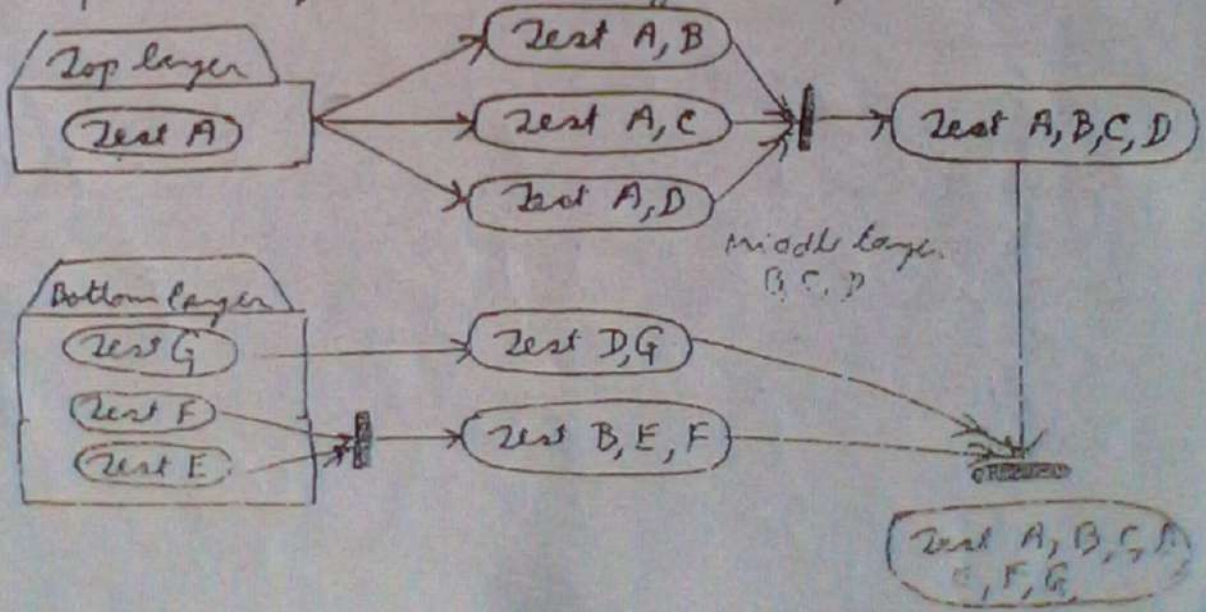
BOTTOM-UP VS. TOP-DOWN TESTING

The advantage of bottom-up testing is that interface faults can be more easily found: when the developers substitute a test driver for a higher-level component, they have a clear model of how the lower-level component works and of the assumptions embedded in its interface. If the higher-level component violates assumptions made in the lower-level component, developers are more likely to find them quickly. The disadvantage of bottom-up testing is that it tests the most important subsystems, namely the components of the user interface, last. First, faults found in the top layer may often lead to changes in the subsystem decomposition or in the subsystem interfaces of lower layers, invalidating previous test drivers and tests. Second, tests of the top layer can be derived from the requirements model and, thus, are more effective in finding faults that are visible to the user.

The advantage of top-down testing is that it starts with user interface components. The same set of tests, derived from the requirements, can be used in testing the increasingly more complex set of subsystems. The disadvantage of top-down testing is that the development of test stubs is time-consuming and prone to error. A large number of stubs is usually required for testing nontrivial systems, especially when the lower level of the system decomposition implements many methods

SANDWICH TESTING

The sandwich testing strategy combines the top-down and bottom-up strategies, attempting to make use of the best of both. During sandwich testing, the tester must be able to reformulate or map the subsystem decomposition into three layers, a target layer ("the meat") a layer above the target layer ("the top slice of the bread") and a layer below the target layer ("the bottom slice of bread"). Using the target layer as the focus of attention, top-down testing and bottom-up testing can now be done in parallel. Top-down integration testing is done by testing the top layer incrementally with the components of the target layer, and bottom-up testing is used for testing the bottom layer incrementally with the components of the target layer. As a result, test stubs and drivers need not be written for the top and bottom layers, because they use the actual components from the target layer.



MODIFIED SANDWICH TESTING

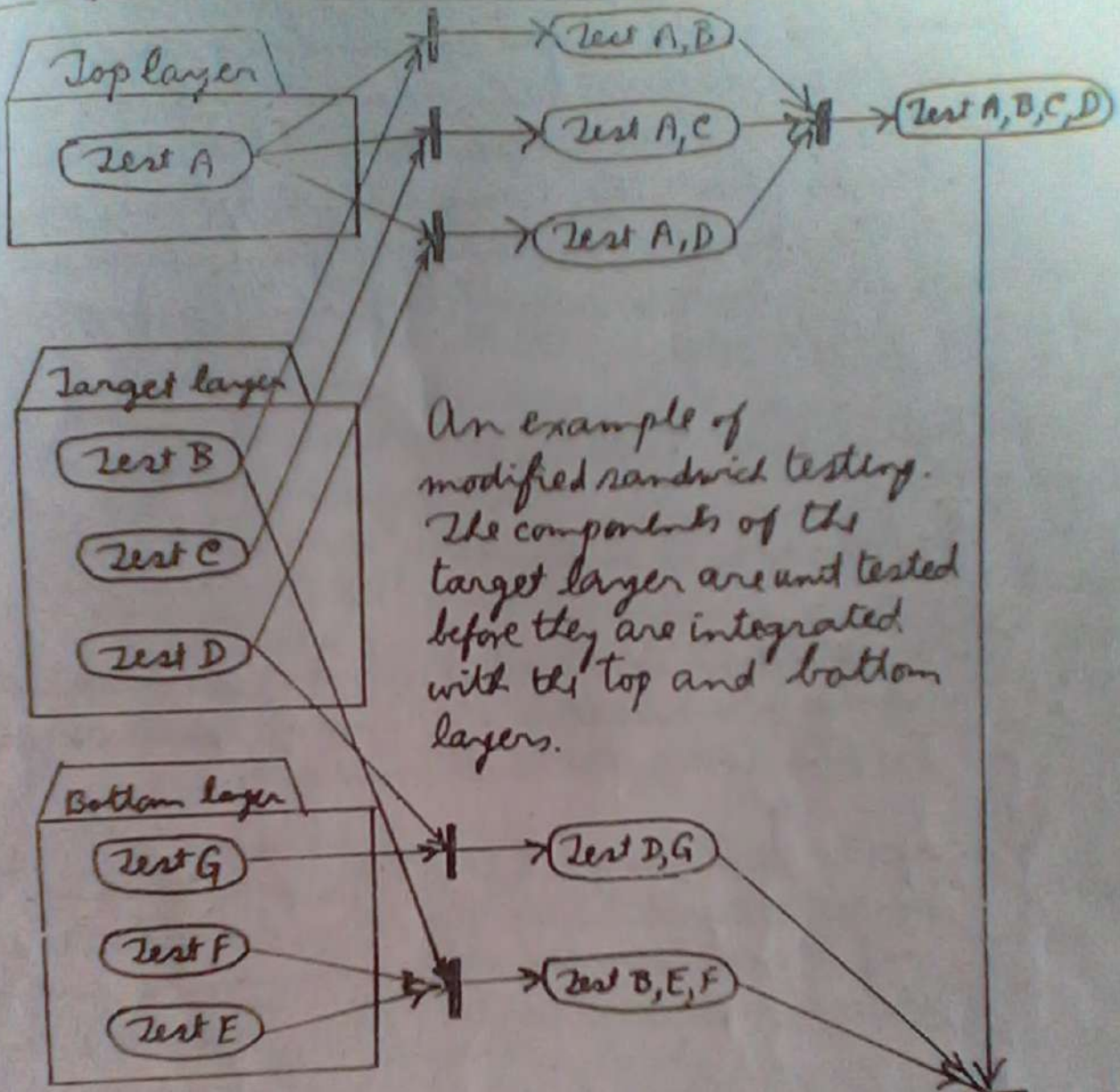
The modified sandwich testing strategy tests the three layers individually before combining them in incremental tests with one another. The individual layer tests consist of a group of three tests:

- a top layer test with stubs for the target layer.
- a target layer test with drivers and stubs replacing the top and bottom layers.
- a bottom layer test with a driver for the target layer.

The combined layer tests consist of two tests:

- The top layer accesses the target layer. The test can reuse the target layer tests from the individual layer tests, replacing the drivers with components from the top layer.
- The bottom layer is accessed by the target layer. This layer can reuse the target layer tests from the individual layer tests, replacing the stubs with components from the bottom layer.

Modified Sandwich Testing (contd.)



An example of modified sandwich testing. The components of the target layer are unit tested before they are integrated with the top and bottom layers.

The advantage of modified sandwich testing is that many testing activities can be performed in parallel. The disadvantage of modified sandwich testing is the need for additional test stubs and drivers. Overall, modified sandwich testing leads to a significantly shorter overall testing time than top-down or bottom-up testing.

System Testing

Unit and integration testing focus on finding faults in individual components and the interfaces between the components. Once components have been integrated, system testing ensures that the complete system complies with the functional and nonfunctional requirements. Several system testing activities are performed:

- Functional testing
- Performance testing
- Pilot testing. Tests of common functionality among a selected group of end users in the target environment.
- Acceptance testing. Usability, functional, and performance tests performed by the customer in the development environment against acceptance criteria (from Project Agreement)
- Installation testing. Usability, functional, and performance tests performed by the customer in the target environment.

System Testing (cont'd)

FUNCTIONAL TESTING, also called requirements testing, finds differences between the functional requirements and the system. System testing is a blackbox technique: test cases are derived from the use case model. In systems with complex functional requirements, it is usually not possible to test all use cases for all valid and invalid inputs. The goal of the tester is to select those tests that are relevant to the user and have a high probability of uncovering a failure.

Functional testing is different from usability testing which also focuses on the use case model. Functional testing finds differences between the use case model and the observed system behaviour, whereas usability testing finds differences between the use case model and the user's expectation of the system.

To identify functional tests, we inspect the use case model and identify use case instances that are likely to cause failures. This is done using blackbox techniques similar to equivalence testing and boundary testing.

PERFORMANCE TESTING finds differences between the design goals selected during system design and the system. Because the design goals are derived from the nonfunctional requirements, the test cases can be derived from the SDD (System Design Document) or from the RAD (Requirements Analysis Document). The following tests are performed during performance testing:

- Stress testing checks if the system can respond to many simultaneous requests. For example, if an information system for car dealers is required to interface with 6000 dealers, the stress test evaluates how the system performs with more than 6000 simultaneous users.
- Volume testing attempts to find faults associated with large amounts of data such as static limits imposed by the data structure or high-complexity algorithms, or high disk fragmentation.
- Security testing attempts to find security faults in the system. There are few systematic methods for finding security faults. Usually this test is accomplished by "tiger teams" who attempt to break into the system, using their experience and knowledge of typical security flaws.
- Timing testing attempts to find behaviours that violate timing constraints described by the nonfunctional requirements.
- Recovery test evaluates the ability of the system to recover from erroneous states such as the unavailability of resources, a hardware failure or a network failure.

Performance testing finds differences between the design goals selected during system design and the system. Because the design goals are derived from the nonfunctional requirements, the test cases can be derived from the SDD (System Design Document) or from the RAD (Requirements Analysis Document). The following tests are performed during performance testing:

- Stress testing checks if the system can respond to many simultaneous requests. For example, in an information system for car dealers is required to interface with 6000 dealers, the stress test evaluates how the system performs with more than 6000 simultaneous users.
- Volume testing attempts to find faults associated with large amounts of data such as static limits imposed by the data structure or high-complexity algorithms, or high disk fragmentation.
- Security testing attempts to find security faults in the system. There are few systematic methods for finding security faults. Usually this test is accomplished by "tiger teams" who attempt to break into the system, using their experience and knowledge of typical security flaws.
- Timing testing attempts to find behaviours that violate timing constraints described by the nonfunctional requirements.
- Recovery test evaluates the ability of the system to recover from erroneous states, such as the unavailability of resources, a hardware failure or a network failure.

System Testing (cont.) PILOT TESTING

During the pilot test, also called the field test, the system is installed and used by a selected set of users. Users exercise the system as if it had been permanently installed. No explicit guidelines or test scenarios are given to the users. Pilot tests are useful when a system is built without a specific set of requirements or without a specific customer in mind. In this case, a group of people is invited to use the system for a limited time and to give their feedback to the developers.

An alpha test is a pilot test with users exercising the system in the development environment. In a beta test the acceptance test is performed by a limited number of end users in the target environment; that is, unlike in usability tests, the behaviour of the end user is not observed and recorded in alpha or beta tests. As a result, beta tests do not test usability requirements as thoroughly as usability tests do. For interactive systems where ease of use is a requirement the usability test cannot be replaced with a beta test.