(3) Although this approach is not overly complicated, in practise it may lead to a variety of practical issues.

(2) <u>Bottom – Up Integration:</u>

* In this instance, start building and testing with the program's lowest level components.
* The processing necessary for components subordinate to a certain level to always be available is made possible by the bottom-up integration of components.
* In this instance, the stub is not required.

Steps followed in the Bottom – Up Integration:

Step 1: Low-level components are assembled to create builds, or clusters, that carry out particular software tasks.

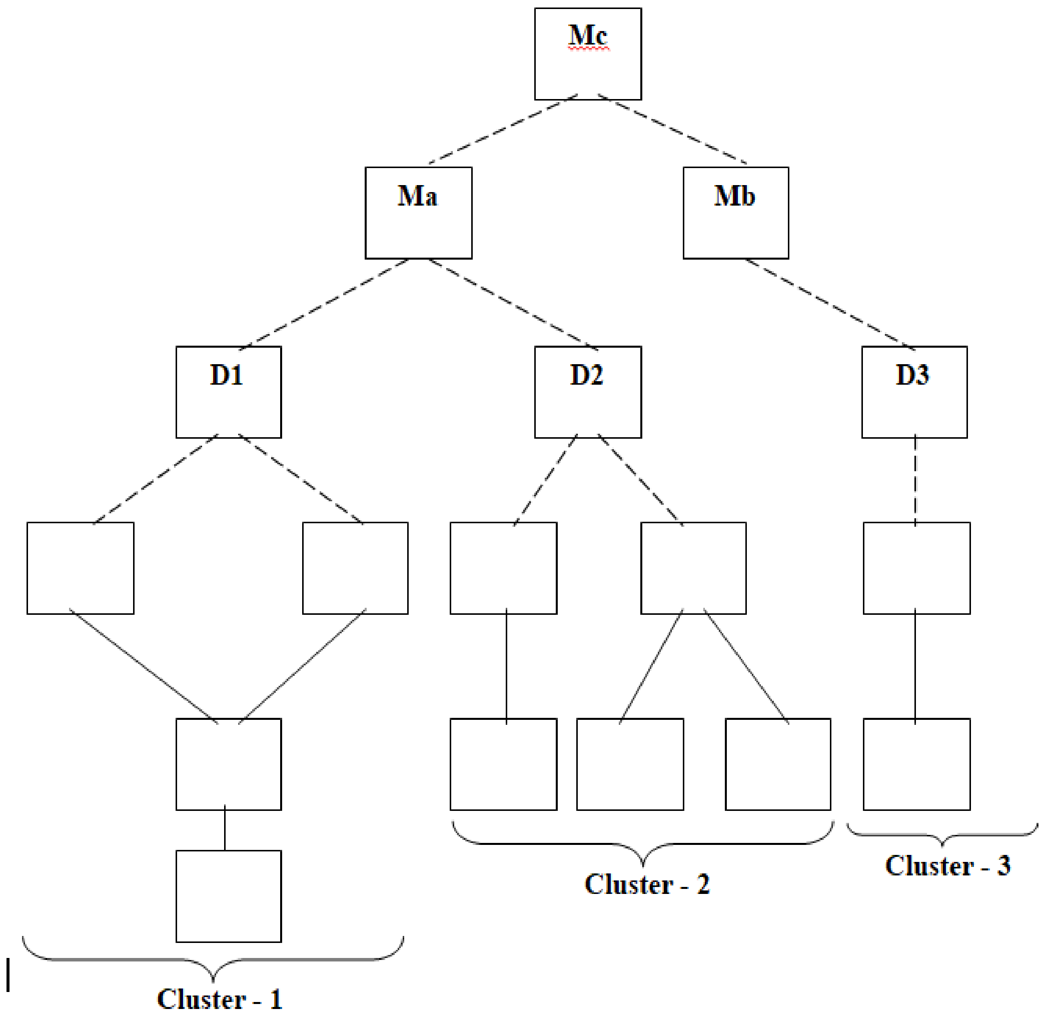
Step 2: Involves writing a driver, or control program for testing, to synchronize the input and output of test cases.

Step 3: Testing the cluster

Step4: Clusters are consolidated and drivers are eliminated as they go up the program structure.

<u>Example:</u>

* Components are assembled to form Clusters 1, 2, and 3 as shown in the image below * Each of the Clusters is put through its paces with the help of a driver * The Ma component takes precedence over the Clusters 1 and 2 components.
* Drivers D1 and D2 are taken out of service, and clusters are now connected directly to Ma.
* In a similar manner, driver D3 for cluster – 3 has been eliminated and has been integrated with MB.
* The components Ma and Mb are both incorporated into the Mc structure.



<u>Bottom up Integration</u>

6.9 <u>Regression Testing:</u>

* It involves rerunning a portion of previously completed tests to make sure that modifications have not had any unanticipated negative effects.

* i.e. whenever software is updated, a portion of the software configuration is altered, including the program, its documentation, and the data that supports it.

* Regression testing is the process used to make sure that modifications made [for testing purposes or for other reasons] don't result in the introduction of new errors.

* Regression testing can be carried out either manually by running a subset of all test cases again or automatically by employing tools for capture and playback.

* Software engineers can record test cases and outcomes with capture and playback tools for later comparison and playback.

* There are three types of test cases in the regression test suite:

(i) A sample set of tests that will run through every feature of the software

(ii) Further testing concentrating on software features that are probably going to be impacted by the modification

(iii) Tests concentrating on the modified software components

<u>Smoke Testing:</u>

* When developing software products, this approach to integration testing is frequently employed.

* It is intended to serve as a patching mechanism for projects that are time-sensitive, enabling the software team to regularly evaluate its work.

Activities included in the smoke testing:

(1) A "Cluster" is assembled from software components that have been converted into code. Every data file, library, reusable module, and engineering component needed to carry out one or more product functions is included in a cluster.

(2) A battery of tests is intended to identify any problems that prevent the cluster from operating as intended.

(3) The product is tested for smoke every day and the clusters are integrated with additional clusters.

* The integration strategy might be either bottom-up or top-down.

<u>Critical Module:</u>

* It is a measure which contains one (Or) more of the following characteristics:

(i) Addresses several software requirements

(ii) Has a high level of control [resides relatively high in program structure]

(iii) is complex (Or) error prone

(iv) Has definite performance requirements

* Testing the crucial module as soon as feasible is recommended.

* Regression tests typically need to concentrate on important module functionalities.

Integration Test Documentation:

* A test specification includes a detailed task description and an overarching plan for software integration.
* This document contains a

=> Test Plan

=> Test Procedure

=> Work product of the software process

* Here, the testing process is broken down into phases and clusters that focus on particular functional and behavioral aspects of the program.

Validation Testing:

* The validation process at the system level concentrates on the following:

=> User – visible actions

=> User recognizable output from the system

* Validation testing is only successful when the program operates as the customer would reasonably expect it to.
* The Software Requirements Specifications define reasonable expectations
* A section of the specification called validation criteria serves as the foundation for a validation testing approach

Validation Test Criteria:

* A test plan specifies the classes of tests to be conducted;
* Test procedures identify individual test cases;
* Software validation is accomplished by a sequence of tests;
* The purpose of the procedures and test plan is to guarantee:

=> Every functional requirement has been met

=> all behavioral characteristics are achieved

=> all performance requirements are attained

=> documentation is correct

=> usability and other requirements are met

* One of the following two scenarios could arise following the validation test:

(i) The function's (or) performance characteristics meet the requirements and are approved.

(ii) A list of deficiencies is made and the derivation from the specification is discovered.

(iii) Derivation (Or) errors found at this point are rarely able to be fixed in time for the delivery date.

Configuration Review (Or) Audit:

* It is a crucial component of the validation procedure.

* The purpose of the review is to make sure that

=> all software configuration elements have been generated or cataloged correctly and

=> contain all the information needed to support each stage of the software life cycle.

Alpha and Beta testing:

* To enable the customer to verify all requirements, a series of acceptance tests are carried out when custom software is developed for a single customer.

* The end-user acceptance test, which is carried out by them instead of software engineers

* Since it is difficult to conduct acceptance tests with every client while developing software for mass use, most software product developers employ a procedure known as alpha (Or) beta testing.

(i) Alpha testing:

* End Users perform it where the developers are located.

* The program is used in a realistic environment;

* The developer is present at all times;

* The developer monitors errors and usage issues;

* Alpha tests are conducted in a closely supervised environment.

.

(ii) Beta testing:

* The beta test is a "Live" implementation of the program in an uncontrollable environment for the developer.
* The end-user logs any issues that come up during beta testing;
* These error reports are forwarded to the developer on a regular basis;
* The software engineer modifies the product based on the error report, and then gets ready to release it to all of the users.
* It is carried out at end-user locations;
* The developer is not present at this time;
* The beta test is a "Live" application of the software in an

System Testing:

* It is a set of several tests with the main goal of thoroughly testing the computer-based system.
* Although the goals of each test vary, they always aim to confirm that the various components of the system have been correctly integrated and are carrying out their designated tasks.
* Despite the fact that this test series' main objective is to thoroughly test the computer-based system

Types:

(i) Recovery Testing
(ii) Security Testing
(iii) Stress Testing
(iv) Performance Testing
(v) Sensitivity Testing

(1) Recovery Testing:

* It is a type of system test that involves purposefully breaking the programme in a number of various ways and checking to see whether or not recovery is carried out properly.
* If the recovery process is automated, often known as being carried out by the system on its own, then

=> * If it is necessary for human intervention to recover the lost data and restart the system, the Mean Time to Repair (MTTR) metric is analysed to evaluate whether or not it falls within the allowed range of values. The evaluation of the validity of the checkpointing processes that occur as a consequence of reinitialization is what ultimately leads to the assessment of data recovery and restarting.

(2) Security Testing:

Good security testing will eventually be able to break into a system if given sufficient time and resources to do so. When designing a system, it is the job of the system designer to ensure that the cost of penetrating the system is higher than the value of the information that would be obtained.

* It ensures that the protective mechanism that is implemented into a system will, in fact, safeguard the system from unauthorised intrusion.

* In the process of evaluating the system's security, the tester acts out the part of a potential threat who wants to break into the system.

* In the process of evaluating the system's security, the tester acts out the part of a potential threat who wants to break into the system.

* It ensures that the safety feature that was incorporated into a

(3) Stress Testing:

* Stress testing is the process of putting a system through its paces in a manner that places an abnormal demand on its resources in terms of quality, frequency, or volume.

Example:

(i) When one (Or) two is the average rate, a unique test that produces the interruptions per second may be created.

(ii) In order to see how input functions react, input data rates may be raised by an order of magnitude.

Test cases requiring the maximum amount of memory (iii) or other resources are run.

(iv) Test cases are created that could result in issues with memory management.

(4) <u>Performance Testing:</u>

* Its purpose is to evaluate software's performance within the framework of an integrated system.

* At every stage of the testing process, performance testing takes place.

* Stress testing and performance testing are commonly coupled, and both hardware and software requirements are usually involved.

* Its purpose is to assess software performance within the framework of a runtime environment.

(5) <u>Sensitivity Testing:</u>

* It's a type of stress testing.

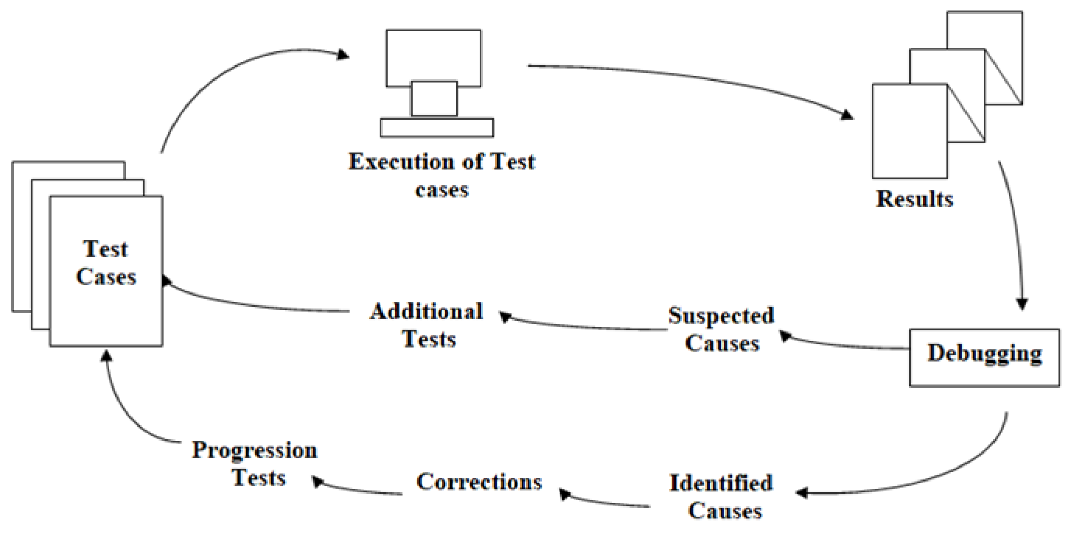
* For instance, a limited set of data that falls inside the boundaries of lawful data may result in inflated and incorrect processing in the great majority of mathematical algorithms.

* Finding data combinations within valid input classes that may cause instability or incorrect processing is required to test for sensitivity.

<u>The Art of Debugging:</u>

* Debugging is not a type of testing, but it does always happen as a consequence of testing. * This is when a test case shows an issue, and debugging is an activity that ultimately leads to the problem being removed from the system.

The Debugging Process



* The execution of a test case is the first step in the debugging process.
Next, the results are analysed, and a discrepancy between expected and actual performance is discovered.
Next, debugging attempts to match symptoms with the underlying causes of errors, which ultimately leads to error correction.
Finally, debugging always has one of two possible outcomes:
(i) the cause will be located and fixed, or
(ii) the cause will not be located. Debugging will always have one of these two outcomes.

Why is debugging so difficult?

(1) The symptom and the cause may be located in different parts of the world [that is, the symptom may appear in one section of a programme, but the reason may actually be situated at a location that is quite far away].
(2) The symptom might go away (temporarily) if another error is fixed.
(3) The symptom might really be caused by something that isn't an error at all (such as rounding off inaccuracy).
(4) The symptom could be the result of an error made by a person that is difficult to pinpoint.
(5) The symptom could be the result of timing problems rather than processing problems.

(6) It may be challenging to correctly duplicate the conditions of the input [for example, in a real-time application when the ordering of the data is unpredictable].
(7) The manifestation of the ailment may come and go. This is especially common in embedded systems, which combine hardware and software in a manner that cannot be separated.
(8) The symptom may be the result of a lot of reasons that are spread across a variety of jobs that are being executed on various processors.

*The amount of pressure to determine the causes of a mistake also grows *This pressure compels the software developer to fix one problem while simultaneously adding two more
*The amount of pressure to determine the causes of a mistake also increases
*The greater the implications of an error, the greater the amount of pressure there is to investigate and pinpoint its root cause.

6.15 Debugging Strategies:

* In general three debugging strategies have been proposed

(i) Brute Force
(ii) Back Tracking
(iii) Cause Elimination

(1) Brute Force:

Memory dumps, runtime traces, and output statements in the code are used for this, and while the resulting data may be useful in the long run, the time and energy spent gathering it are often better spent elsewhere.

* this method is applied only when all other methods fail.
* the philosophy used here may be "Let the computer finds the error."
* we apply this method only when all other methods fail.
* we only apply this method when all other methods fail.
* we only apply this method when all other methods fail.

(2) Back Tracking:

* This method is the most frequent one that can be implemented effectively in relatively modest programmes.

* The source code is traced backward [manually] starting at the spot where a symptom has been detected and continuing until the reason is found.

* If the number of source lines increases, the number of possible backward paths may become unmanageably large.

* The number of possible backward paths could grow uncontrollably if the number of source lines rises.

* The number of possible backward paths could become unmanageable if the number of source lines rises.

* The number of alternative backward paths may grow unmanageably enormous if the source code contains a big number of lines.

* If the source code contains a large number of lines, the number

(3) Cause Elimination:

* This can be accomplished through the use of induction (or) deduction

* It presents the idea of binary partitioning. It is followed by the following steps:

* A cause hypothesis is formulated and is supported (or refuted) by the data pertaining to the mistake occurrence.

* Every potential reason is enumerated, and experiments are carried out to rule them all out.

* When preliminary investigations suggest that a specific cause theory holds potential, data is reworked to try and find the bug.

* A binary partitioning scheme is introduced

* Testing is done to rule out each potential reason after a list of all potential causes is created.

6.16 White Box Testing:

* Another name for this test is the Glass-Box Test

Testing in the white box environment gives the software developer the ability to design test cases that:

(1) Guarantee that all independent routes contained within a module have been investigated at least once;

(2) Exercise all logical judgements on both their true and false sides.

(3) Ensure that all loops are executed at their respective boundaries and within the scope of their respective operational bounds; and

(4). Perform tests on the organization's internal data structures to confirm that they are correct.

White box testing

• To test logical paths across the software, test cases with particular sets of conditions and/or loops are provided.

• Testing of the software is referred to as white-box testing when it is dependent on a detailed investigation of the product's procedures.

• The "status of the programme" can be evaluated at a number of different periods in time.

• White-box testing, also known as glass-box testing, is a method for designing test cases that derives test cases by using the control structure of the procedural design. This method is frequently referred to as "glass-box testing."

Through the use of this strategy, SE is able to generate test cases that

1. Ensure that each and every independent branch contained within a module has been traversed at least once.

2. Consider all logical judgements from both the correct and incorrect perspectives,

3. Ensure that all loops are executed at their respective limits and remain inside their respective operational limitations

4. Perform tests on the internal data structures to confirm that they are valid.

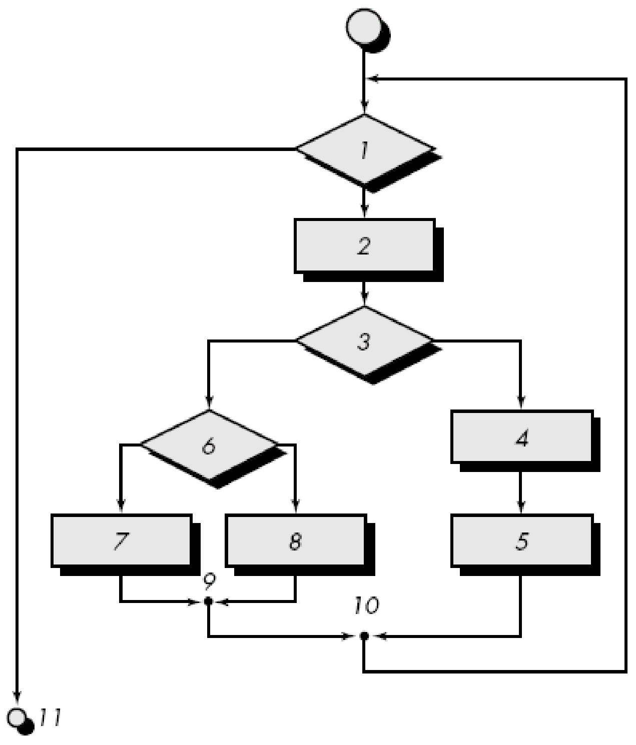
Basis path testing

• Test cases with specific sets of criteria and/or loops are supplied in order to evaluate logical paths throughout the program.

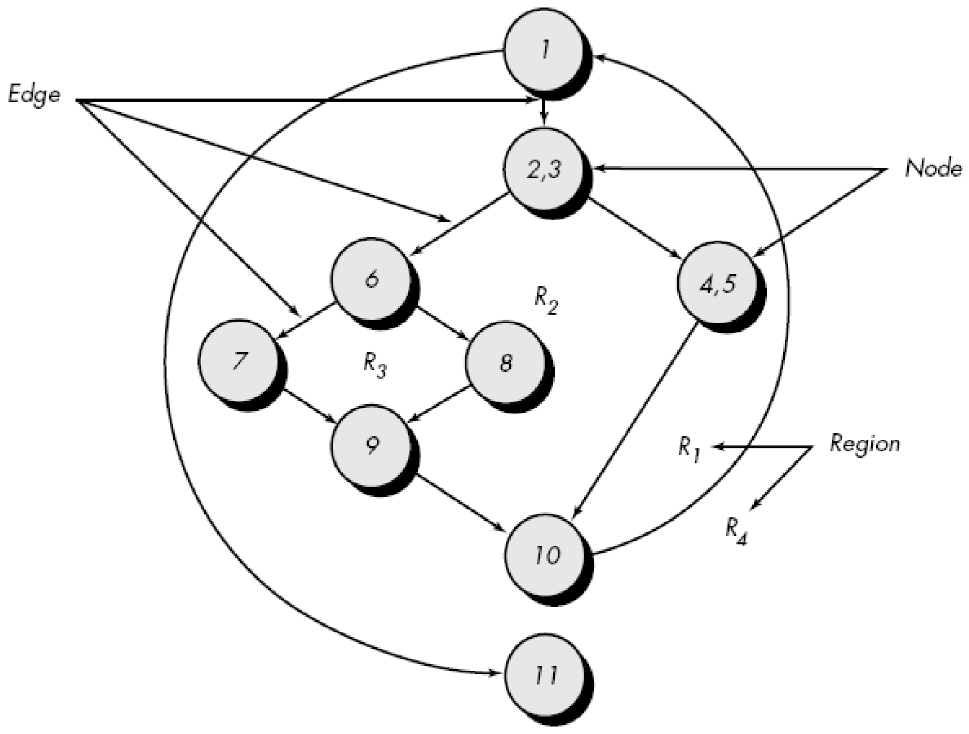
• It is assured that each statement in the programme will be run at least once when using test cases that are derived to exercise the basis set.

Methods:

1. Flow graph notation
2. Independent program paths or Cyclomatic complexity
3. Deriving test cases
4. Graph Matrices



(A)



(B)

Each circle in figure B, which is referred to as a node on a flow graph, represents one or more procedural statements.

• A mapping into a single node is possible for a series of process boxes and a decision diamond.

• Similar to the arrows on flowcharts, the arrows on the flow graph, sometimes referred to as edges or links, show the flow of control.

• Even in cases when a node does not reflect a procedural statement, it still needs an edge to finish at that node.

• Any area bounded by a network of edges and nodes is referred to as a region. The area outside the graph is counted together with the regions when we calculate the overall count.

• In the event that a compound condition arises during the process of procedural design, the flow graph will become marginally more convoluted.

Independent program paths or Cyclomatic complexity

• An independent path is any way through the programme that includes at least one new set of processing statements or new condition. This is the minimum requirement for a path to be considered independent.

• Take, for instance, the following as an illustration of a set of independent paths for a flow graph:

1-11 is the first path, and

1-2-3-4-5-10-1-11 is the second.

– Path 3: 1-2-3-6-8-9-1-11

– Path 4: 1-2-3-6-7-9-1-11

• Take note that every new path results in the creation of a new edge.

• The path 1-2-3-4-5-10-1-2-3-6-8-9-1-11 is not an independent path because it is merely a combination of paths that have already been stated and does not traverse any new edges. This means that it does not go through any new nodes.

• Test cases should be built in such a way that they are forced to follow these basic set paths.

• Each and every line in the programme should have at least one opportunity to be run, and each and every condition should have been tested both ways (true and false).

How can we determine the total number of possible routes to investigate?

• Cyclomatic complexity is a software metric that provides a quantitative measure of the logical difficulty of a programme. It does this by counting the number of paths through a programme.

•It offers the number of tests that need to be carried out in addition to defining the total number of independent pathways in the basis set.

•Calculating cyclomatic complexity can be done one of three ways:

1. The cyclomatic complexity is directly proportional to the number of areas.

2. The cyclomatic complexity, denoted by the symbol $V(G)$, of a flow graph, G, is defined as $V(G) = E - N + 2$, where E represents the number of edges in the flow graph and N represents the number of nodes in the flow graph.

3. A flow graph's cyclomatic complexity, $V(G)$, can be written as $V(G) = P + 1$, where P is the total number of nodes and edges in the predicate.

As a result, we have an upper bound on the total number of tests based on the value of $V(G)$.

Deriving Test Cases

• This strategy consists of a predetermined order of actions.

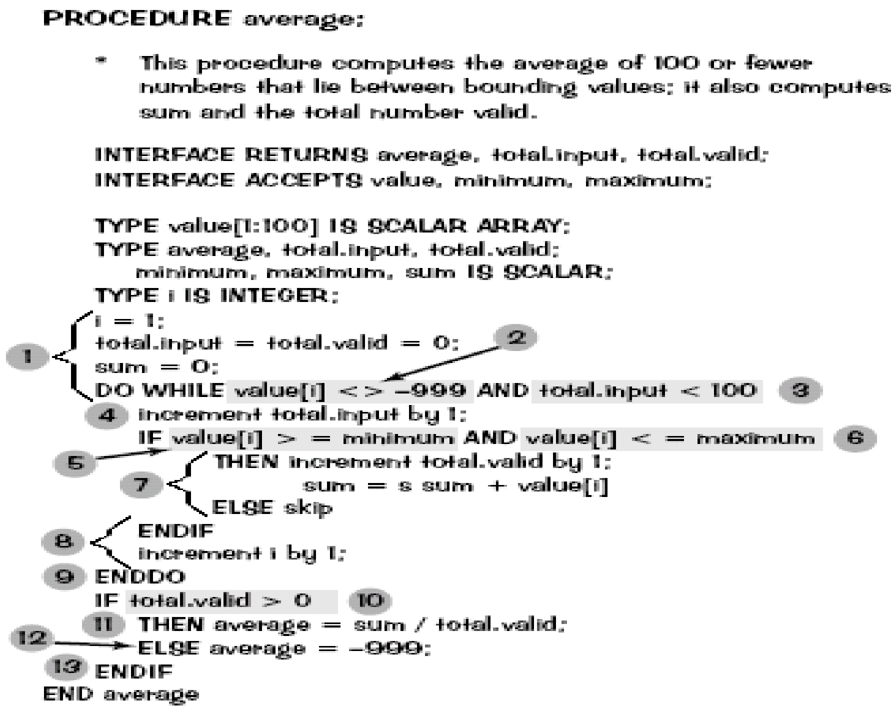
• The average of the operation as shown in the PDL.

• The extremely straightforward algorithm known as average includes both complex criteria and loops.

To derive basis set, follow the steps.

1) A flowchart should be created based on the design or code.

Numbering the PDL statements that will become nodes in the flow graph is the first step in making the graph.

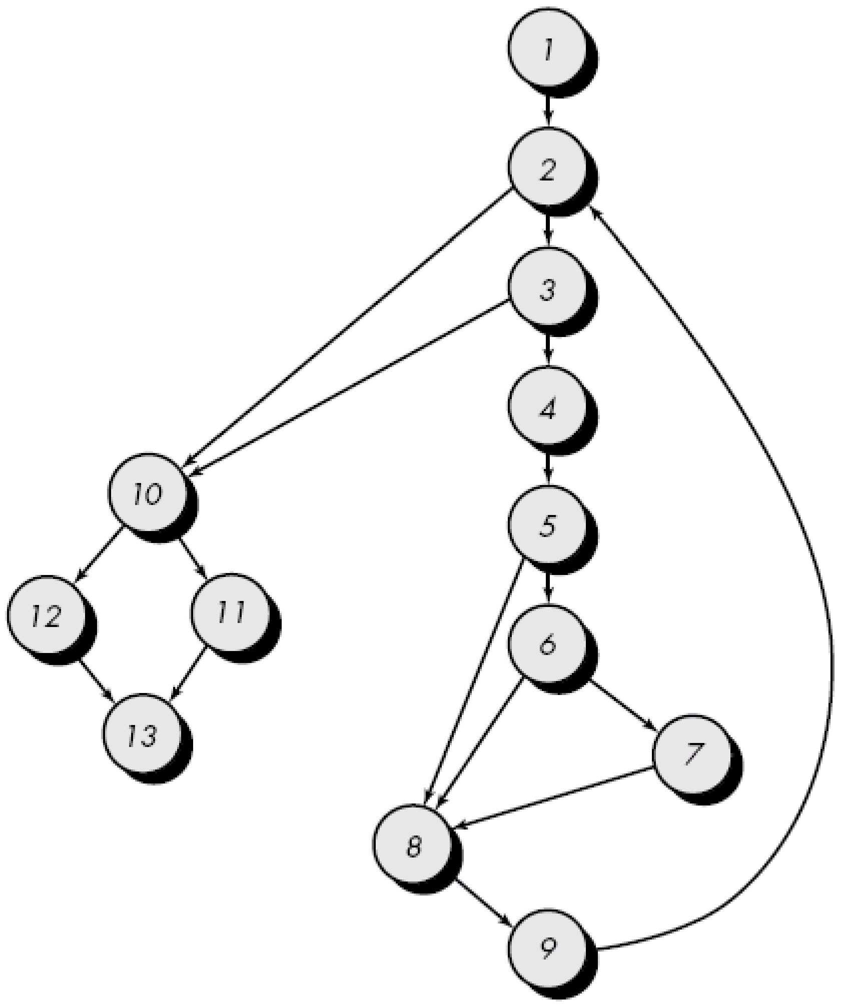


```
PROCEDURE average;

    *   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;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100
       increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum
          THEN increment total.valid by 1;
               sum = s sum + value[i]
          ELSE skip
       ENDIF
       increment i by 1;
    ENDDO
    IF total.valid > 0
       THEN average = sum / total.valid;
       ELSE average = -999;
    ENDIF
END average
```



Flow graph for the procedure average

Determine the cyclomatic complexity of the resultant flow graph.

1. *Second, V(G) can be calculated without making a flowchart by counting the number of conditional statements in the PDL (complex conditions count as two for the average method) and then adding 1.*
2. *Theorem 3: V(G) = 6 regions*
3. *Fourth, V(G) = 17 - 13 + 2 = 6*
4. *In other words, V(G) = 5 predicate nodes+ 1 = 6*

Determine a basis set of linearly independent paths

a. The value of *V*(*G*) provides the number of linearly independent paths through the program control structure.
b. path 1: 1-2-10-11-13
c. path 2: 1-2-10-12-13

   d. path 3: 1-2-3-10-11-13
   e. path 4: 1-2-3-4-5-8-9-2-. . .
   f. path 5: 1-2-3-4-5-6-8-9-2-. . .
   g. path 6: 1-2-3-4-5-6-7-8-9-2-. . .
   h. The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

5. **Prepare test cases that will force execution of each path in the basis set.**
   a. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
   b. Each test case is executed and compared to expected results.
   c. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

<u>Black – Box Testing:</u>

Black box testing is a complementary strategy that is likely to reveal a different class of problems than white-box testing methods; it is not meant to be used in place of white box testing.

* It is also known as Behavioural Testing.

* White box testing and black box testing are not interchangeable.

* Black box testing is also known as Behavioural Testing. Testing in a black box makes an effort to identify problems in the following areas:

(1) Incorrect Functions, Or Functions That Are Missing

(2) Errors in the interface

(3) Errors in the data structures (Or) unauthorised access to external data bases

(4) Inappropriate behaviours (Or) Poor performances

(5) Errors during the initialization and termination processes

* By applying testing methodologies known as black box testing, we are able to develop a set of test cases that are compliant with the following criteria.(i) Test cases that reduce the number of additional test cases that need to be written to ensure reasonable testing by a count that is greater than one (ii) Test cases that provide information on the presence (Or) absence of classes of mistakes rather than errors that are just associated with the particular test that is now being carried out (iii) Test cases that provide information on the

presence (Or) absence of classes of mistakes rather than errors that are just associated with the particular test that is now being carried out (iv) Test cases that

• Testing using a black box whereas the control structure is being willfully disregarded, the emphasis will be placed on the information sphere. Exams are structured to provide responses to the following questions:
Testing for functional validity entails what steps, exactly?
How are the system's performance and behaviour put to the test?
Which types of input will make the most effective test cases?
• We are able to build a set of test cases that are in agreement with the following criteria thanks to the utilisation of black-box approaches, which are described below.
- Test cases that cut down on the number of additional test cases that have to be designed in order to conduct testing that is regarded to be reasonable (thus minimising the amount of work that must be done and the amount of time that must be spent).

— Test cases that provide information regarding the existence or non-existence of specific sorts of faults.

- Testing procedures using black boxes
- Methods of Testing Relying on Graphs
- Partitioning based on equivalence
- Boundary value analysis, often known as BVA
- Orthogonal Array Testing

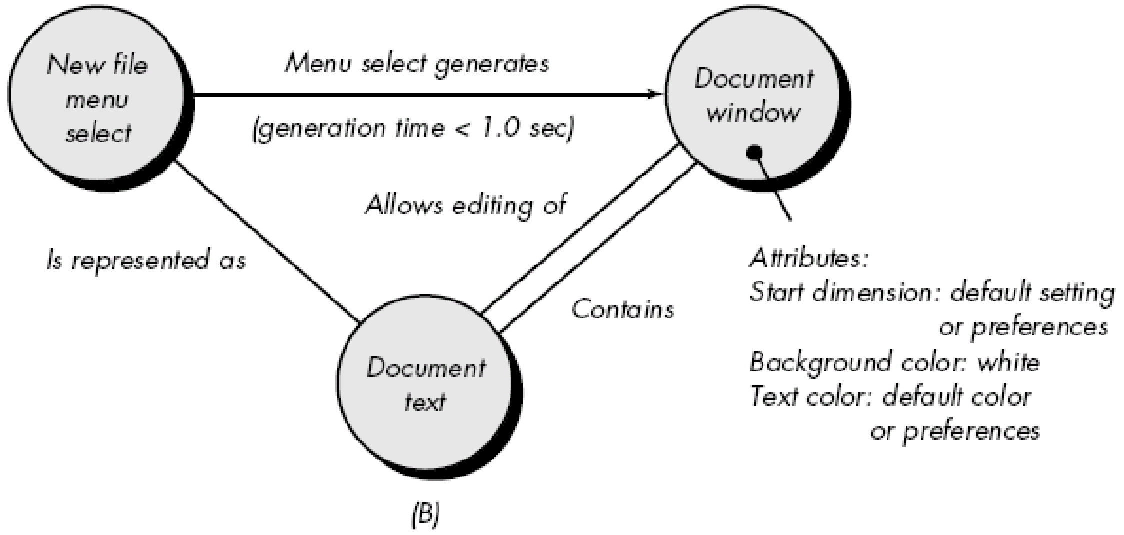
Graph-Based Testing Methods

• To have an understanding of the relationships that connect the items that are modelled in software as well as the objects themselves.
• The following step is to develop a series of tests that will verify that "all objects have the expected relationship to one another."
•Alternately stated as follows:
- Make a diagram depicting significant things and the connections between them.
- Conceive a battery of examinations that will be centred on the graph.
• In order to ensure that every item and relationship is tested and that any errors are located.

• To begin, design a graph that consists of a set of nodes, which stand for items, and links, which depict the connections that exist between those objects. The nodes will represent the items, and the links will show the relationships between the items.
– weights that describe some aspect of a link, which are referred to as link weights; – weights that characterise the qualities of a node, which are referred to as node weights.

• The nodes in the network are depicted as circles, and the connections between them can take on a variety of forms.
• A one-way relationship is denoted by a directed link, which is depicted as an arrow and indicates that the link only goes in one direction.
• The relationship is considered to apply in both directions when there is a bidirectional link, which is also referred to as a symmetric link.
When multiple distinct associations need to be constructed between two nodes in a graph, the use of parallel links is necessary.

Example



• Object #1 is a selection from the new file menu
• Object #2 is a window for the document
• Object No. 3 is the text of the document

A document window is produced when a new file is selected from the menu (see the illustration to the right).

• According to the link weight, the window must be formed in fewer than one hundred sixty five milliseconds.

• The document window's node weight gives a list of the window properties that should be anticipated when the window is formed. This list can be found in the document window.

• A parallel connection indicates a relationship between the document window and the document text. • An undirected link creates a symmetric relationship between the new file menu select and the document text.

To define equivalence classes follow the guideline

1. If the input condition requires the specification of a range, then one valid and two invalid equivalence classes are defined.
2. There is one defined legitimate equivalency class and two defined invalid equivalency classes when an input condition requests a certain value.
3. An equivalence class will be developed for the case in which an input condition identifies a member of a set.
4. A valid class and an invalid class are specified for the case where an input condition is Boolean.

Example

• the area code, which might be a blank number or a three-digit number

• prefix, a three-digit number that does not start with zero or one

• suffix, sometimes known as a four-digit number

• Password: an alphabetic and numeric string consisting of six digits

• commands such as cheque, deposit, and bill pay, as well as similar ones

• Input condition, Boolean—the area code may or may not be present; the condition can either be true or false.

• Please provide the condition and value as a three-digit number.

• prefix: • Input condition, range—values defined between 200 and 999, with certain exceptions to the rule.

• Input condition, value—four-digit length • Suffix: • Input condition, value

• password: • Input condition, Boolean — the presence of a password can either be present or absent.

• Input condition, value, is a string of six characters.

• command: • input condition, set— check, deposit, and bill pay.

Boundary Value Analysis (BVA)

- Equivalence partitioning is a technique for the creation of test cases, while boundary value analysis is a technique that complements it.
- • The BVA method will lead to the selection of test cases that are located on the "edges" of the equivalence class, as opposed to selecting any element of the class.
- • To put it another way, instead of concentrating simply on the circumstances of the input domain, BVA generates test cases from the output domain as well.

Guidelines for BVA

1. If an input condition specifies a range that is bounded by the values a and b, test cases should be built using the values a and b, as well as the values immediately above and immediately below a and b.
2. If an input condition requires a certain number of values, test cases should be designed that exercise both the lowest possible value and the highest possible value. Additionally tested are values that lie just above and below the minimum and maximum, respectively.
3. Apply the first and second guidelines to the output conditions.
4. If the internal programme data structures have borders that have been mandated, make sure to construct a test case that will exercise the data structure at its boundary.

Orthogonal Array Testing

- There are not many input parameters, and the possible range of values that each parameter can potentially take is clearly specified and constrained.
- When these numbers are relatively tiny, it is conceivable to consider every potential input permutation. One example of this would be having three input parameters, each of which might take on three discrete values.
- • On the other hand, exhaustive testing is required when the number of input values increases along with the number of discrete values for each data item. • A method known as orthogonal array testing is one that can be utilised in situations where exhaustive testing is not possible due to the problem's relatively large scope but where the input domain is relatively limited.