Coding and Testing

Coding

- Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously.
- The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:
  - **A coding standard gives a uniform appearance to the codes written by different engineers.**
  - **It enhances code understanding.**
  - **encourages good programming practices.**

Coding standards and guidelines

- **Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.**

The following are some representative coding standards.

1. **Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.**
2. **Contents of the headers preceding codes for different modules: The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:**

• Name of the module.
• Date on which the module was created.
• Author's name.
• Modification history.
• Synopsis of the module.
• Different functions supported, along with their input/output parameters.
• Global variables accessed/modified by the module.

1. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

1. **Error return conventions and exception handling mechanisms:** The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

The following are some representative coding guidelines recommended by many software development organizations.

1. **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.

1. **Avoid obscure side effects:** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

1. **Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result.
2. **The code should be well-documented:** As a rule of thumb, there must be at least one comment line on the average for every three-source line.
3. **The length of any function should not exceed 10 source lines:** A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
4. **Do not use goto statements:** Use of goto statements makes a program unstructured and makes it very difficult to understand.

Code review

- Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module.

1. **Code Walk Through:**
   - Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated, a few members of the development team are given the code few days before the walk through meeting to read and understand code.

The main objectives of the walk through are to discover the algorithmic and logical errors in the code.

- **Guidelines:**

- The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

1. Code Inspection:

- the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
- software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Non terminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation.
- Mismatches between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equally of floating point variables, etc.

Code Walk Through:

It discover the algorithmic and logical errors in the code.

Code Inspection

It checks the presence of some common type errors that creep into code due to programmer mistakes and oversight and to check Whether coding standards have been adhered to.

Software documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Use documents help the users in effectively using the system.
- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

Different types of software documents

It is broadly classified into the followings:

1. Internal documentation:
   - Internal documentation is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.
   - Careful experiments suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code.

1. **External documentation:**

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

Gunning's fog index:

Metric used to measure the readability of a document.

Fog(D)= 0.4 x (words/sentences) + percent of words having 3 or more syllables

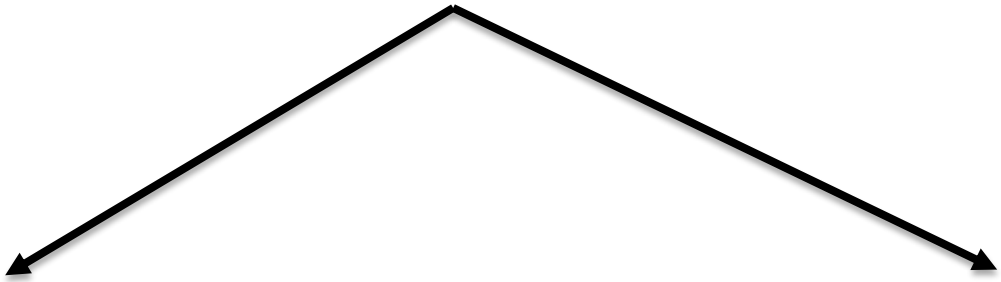
"The Gunning fog index is based on the premises that use of short sentences and simple words makes a document easy to understand"

Fog(D)= 0.4 (23/1)+(4/23)*100=25

If user educational qualification is Class X, then gunning's fog index does not exceed X.

Testing

Finds



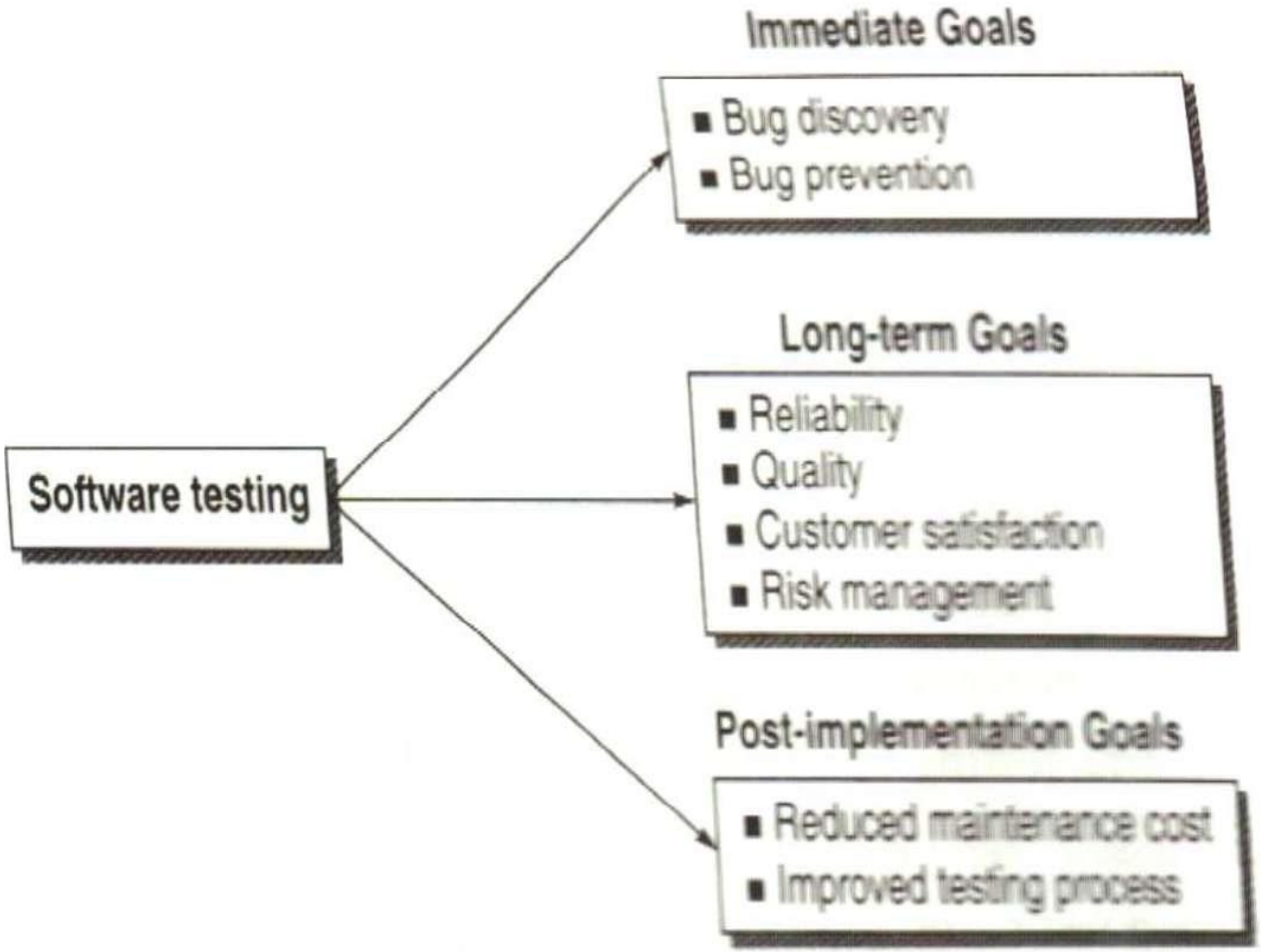
Absence of Errors

Presence of Errors

Aim of Testing

- The aim of the testing process is to identify all defects **existing** in a software product.
- However for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large.
- It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product.
- Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Testing Goals



Program Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

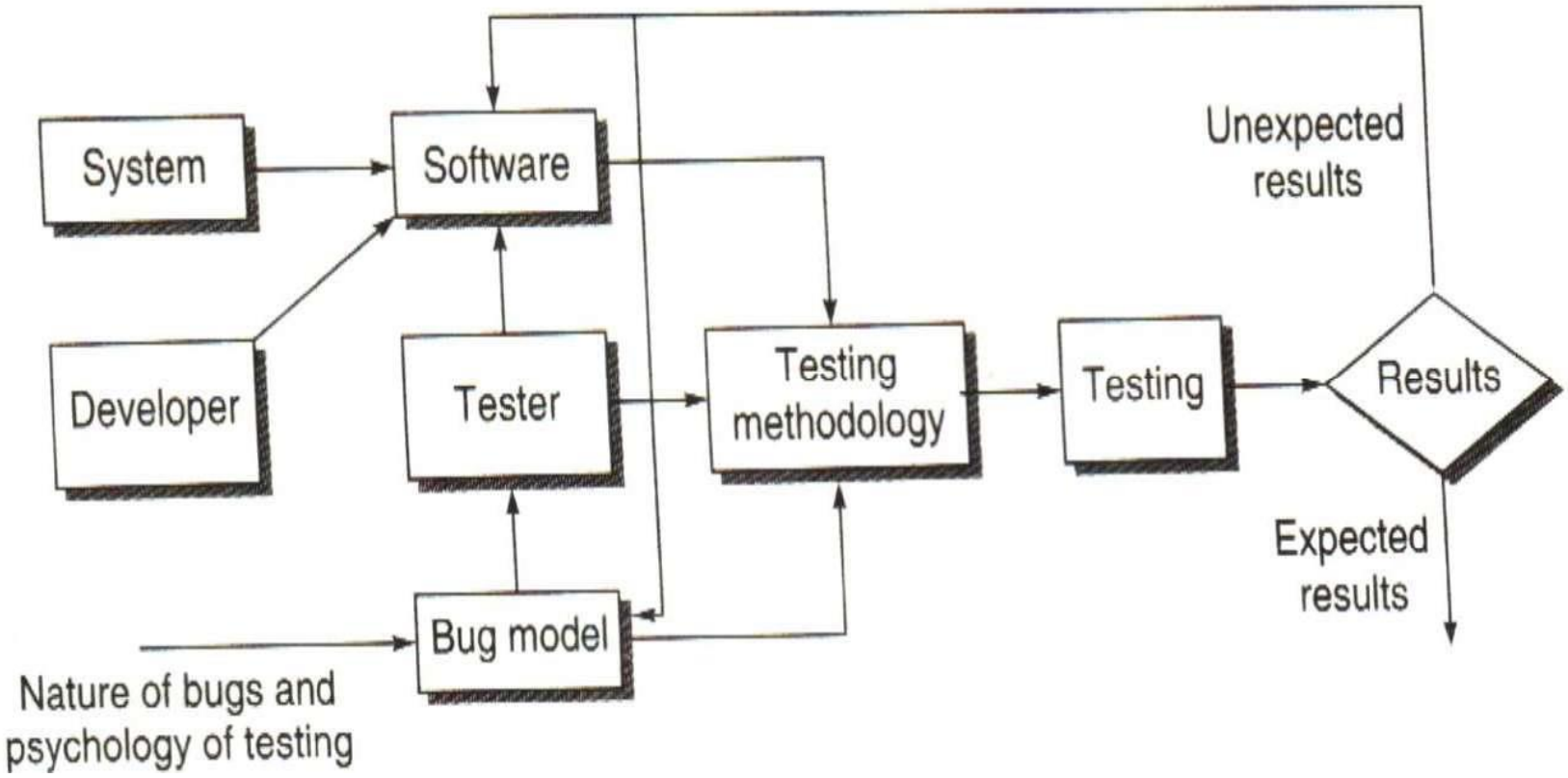
- **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.
- **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.
- **Test suite:** This is the set of all test cases with which a given software product is to be tested.

Differentiate between verification and validation

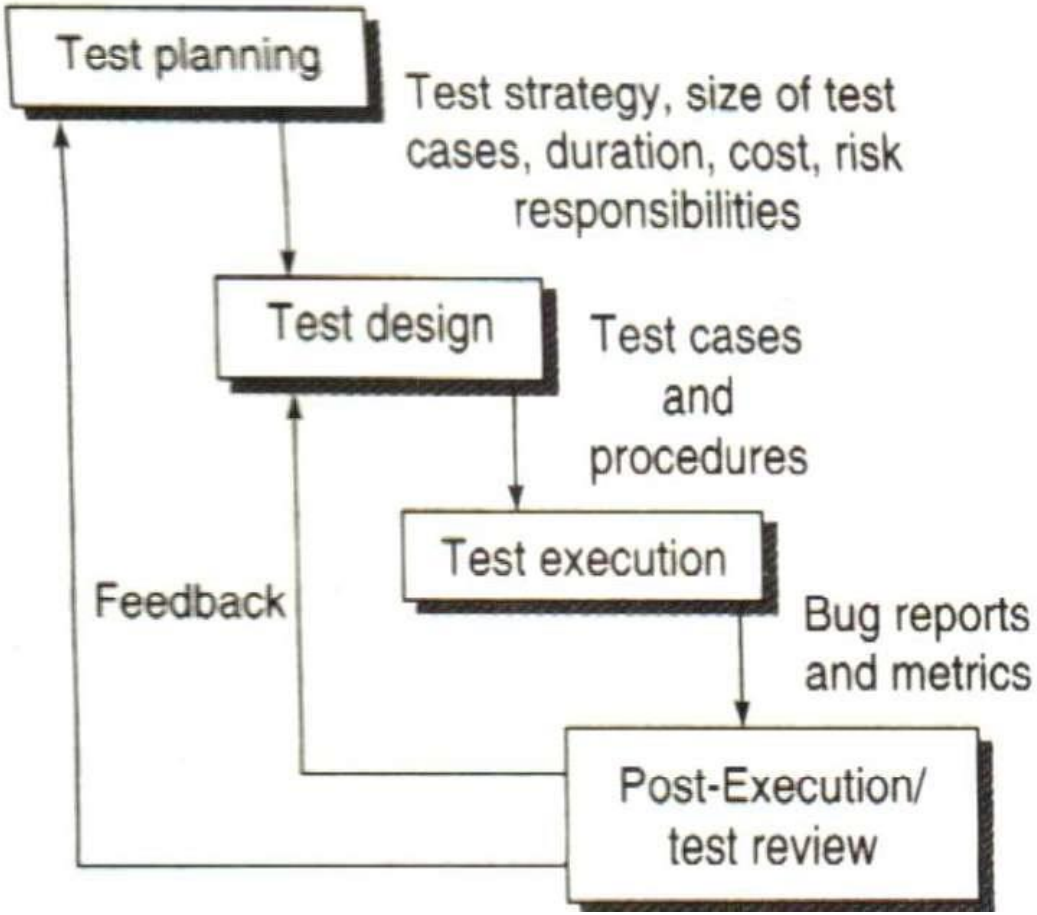
- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase.

- Validation is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

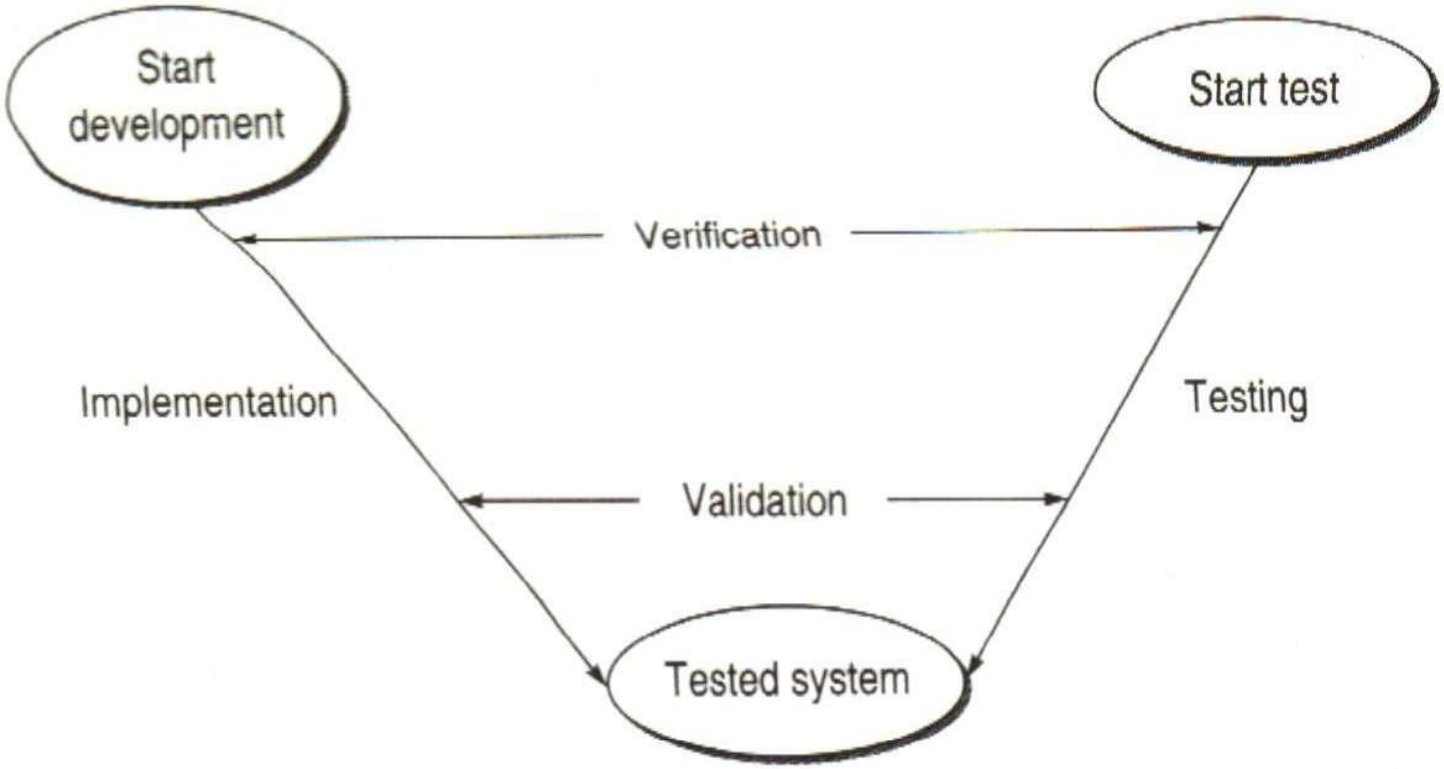
Testing Model



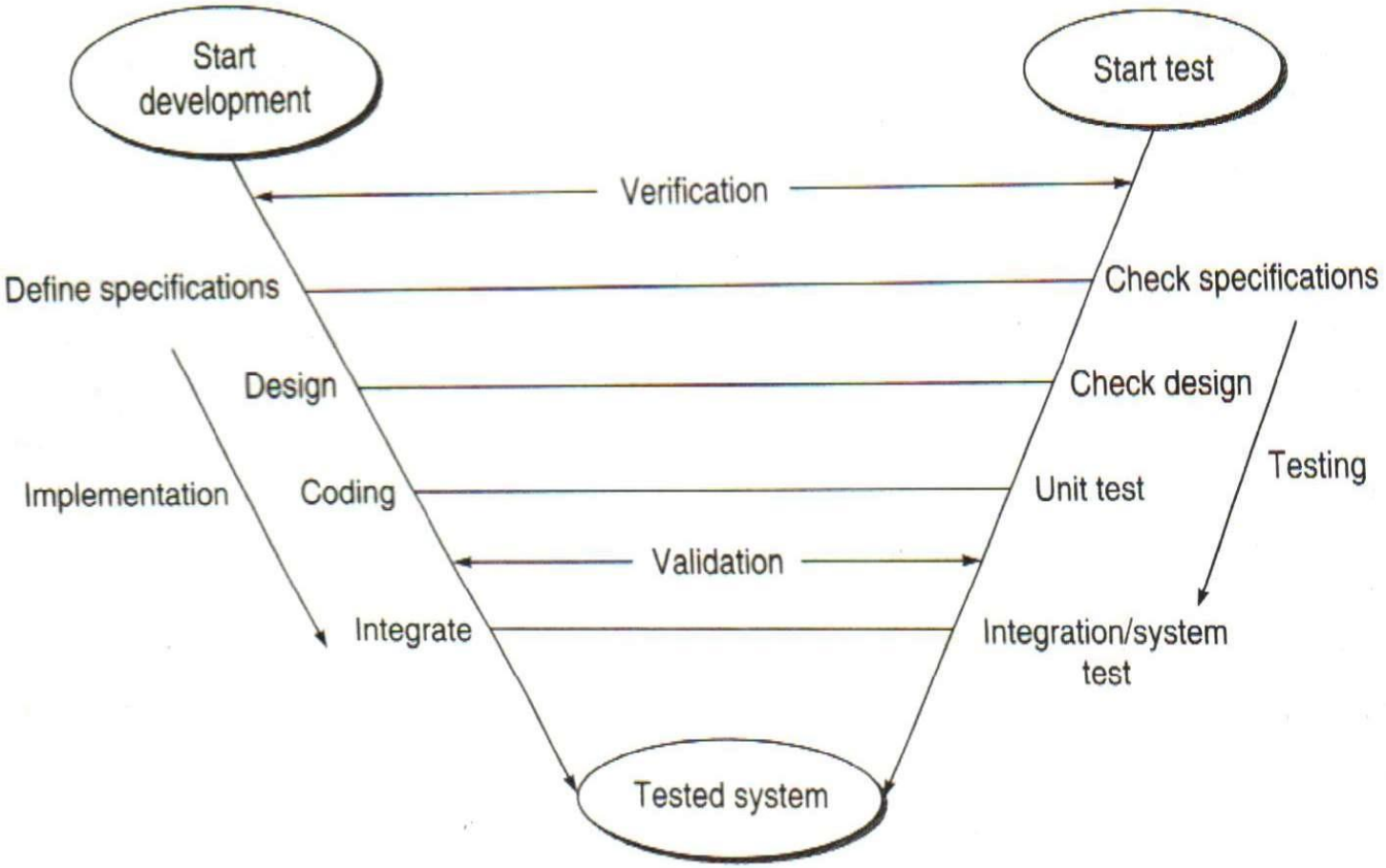
STLC


Test planning
Test strategy, size of test cases, duration, cost, risk responsibilities
Test design
Test cases and procedures
Test execution
Bug reports and metrics
Post-Execution/ test review
Feedback

V-shaped Test model



V-shaped Test model



Functional testing vs. Structural testing

- In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing.

- On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

Black-Box Testing

Testing in the large vs. testing in the small

- Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

Unit testing

- Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

Black box testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required. The following are the two main approaches to designing black box test cases.

- **Equivalence class Partitioning**
- **Boundary value analysis**

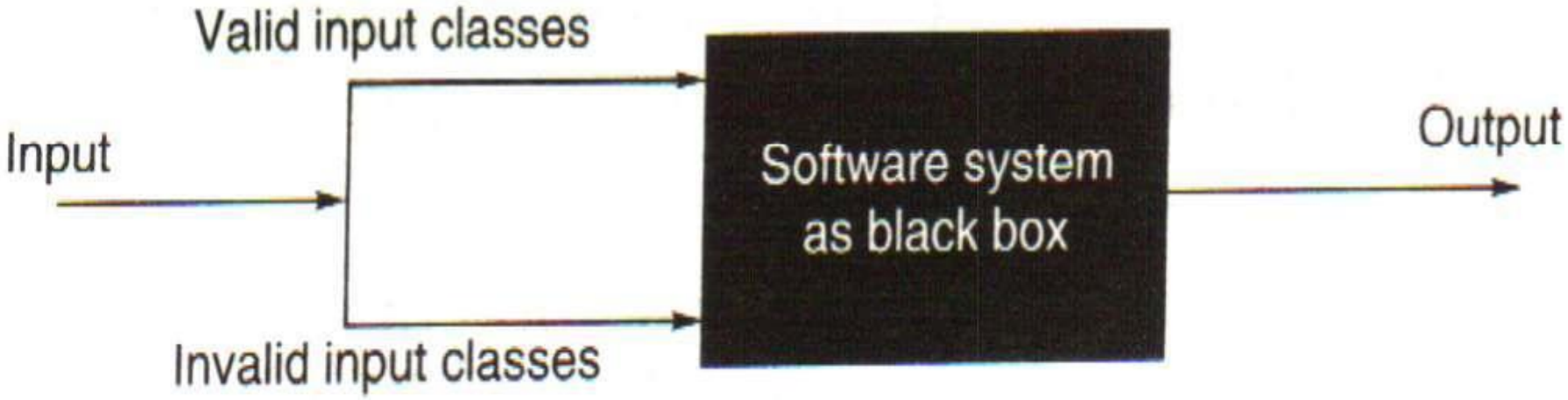
Equivalence class Partitioning

- In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class.
- The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class.
- Equivalence classes for a software can be designed by examining the input data and output data.

The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Equivalence class testing



Example#1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

The equivalence classes are the following:

- Parallel lines (m1=m2, c1≠c2)
- Intersecting lines (m1≠m2)
- Coincident lines (m1=m2, c1=c2)

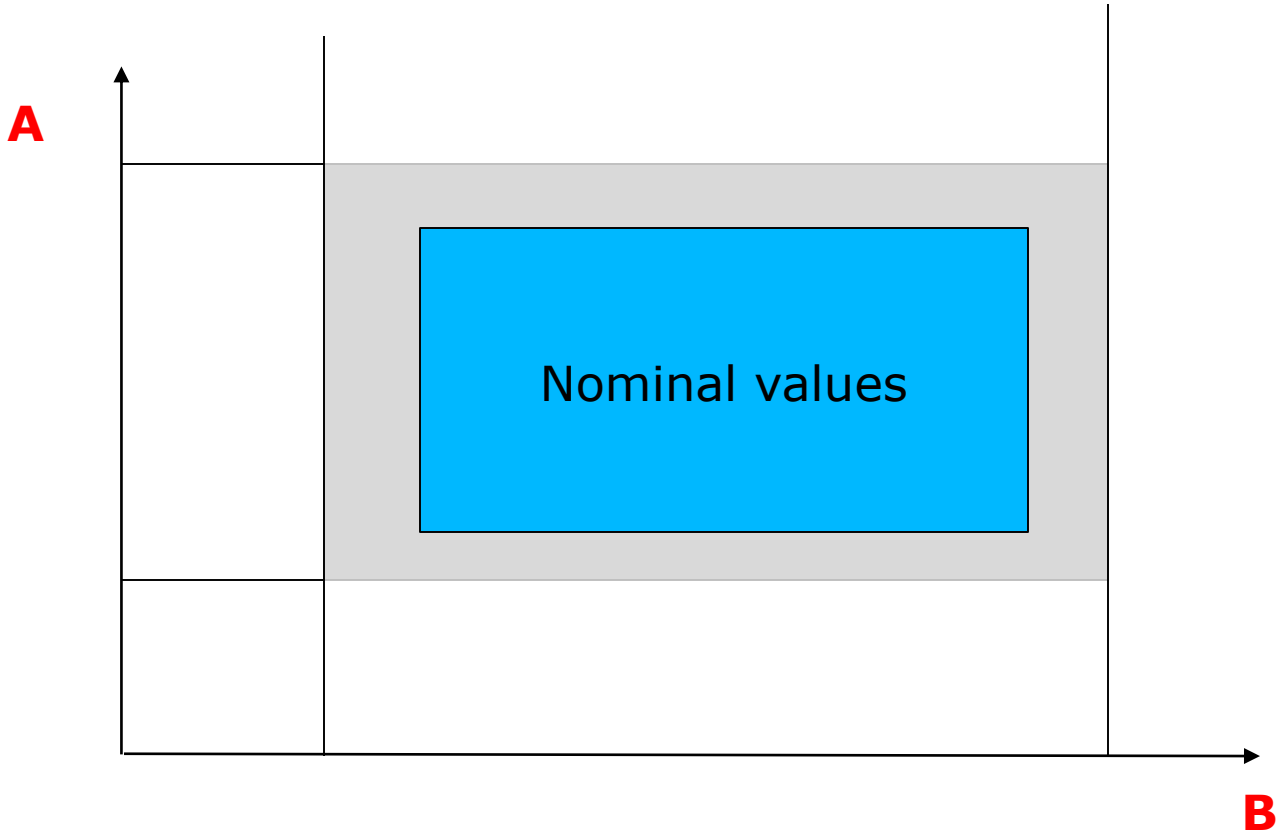
Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

Boundary Value Analysis

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1,5000,5001}.

Boundary Value Analysis



Boundary Value Analysis

1. **Boundary Value Checking (BVC):** Consider One variable at its extreme and other variables at their nominal values in the input domain.

The variable at the extreme can be selected at: Min, Min+, Max, Max-

For n variable in a module, **4n+1** test cases can be designed.

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

Boundary Value Analysis

2. Robustness Testing Method: Consider that boundary values that are exceeded Min and Max value Like (Max+, Min-)

The variable at the extreme can be selected at: Min, Min+, Max, Max- . Now append Max+, Min-

Add the following cases with other previous cases

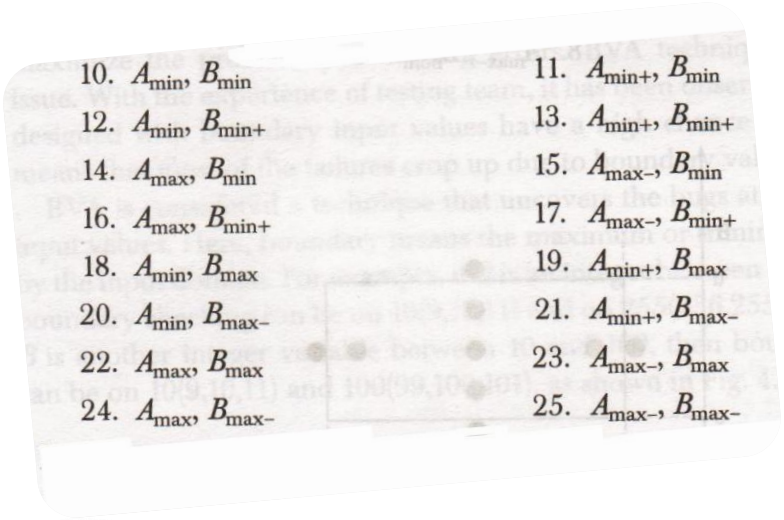
10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}

For n variable in a module, **6n+1** test cases can be designed.

Boundary Value Analysis

3. Worst case Testing Method: Consider more than one variable (BVC) at its extreme and other variables at their nominal values in the input domain.

The variable at the extreme can be selected at: Min, Min+, Max, Max-



10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}
14. A_{max}, B_{min}
15. A_{max-}, B_{min}
16. A_{max}, B_{min+}
17. A_{max-}, B_{min+}
18. A_{min}, B_{max}
19. A_{min+}, B_{max}
20. A_{min}, B_{max-}
21. A_{min+}, B_{max-}
22. A_{max}, B_{max}
23. A_{max-}, B_{max}
24. A_{max}, B_{max-}
25. A_{max-}, B_{max-}

For n input variable in a module, 5^n test cases can be designed.

Problem: A program reads an integer number within the range [1,100] and determines whether it is prime number or not. Design test Cases for this program using BVC, robust testing and worst-case testing methods.

BVC: 1 variable so total cases will be 4n+1=5

Min value = 1
Min^{+} value = 2
Max value = 100
Max^{-} value = 99
Nominal value = 50–55

In worst case testing, total number of cases are also 5^1 = 5, It is same as BVC

Boundary Value Analysis

BVC: 1 variable so total cases will be 4n+1=5

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Boundary Value Analysis

Robust Testing: 1 variable so total cases will be 6n+1=7

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

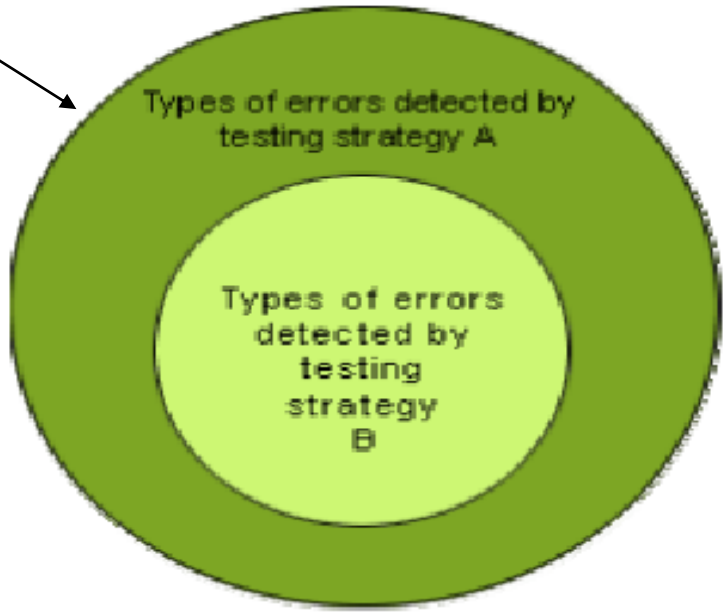
Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

White-Box Testing

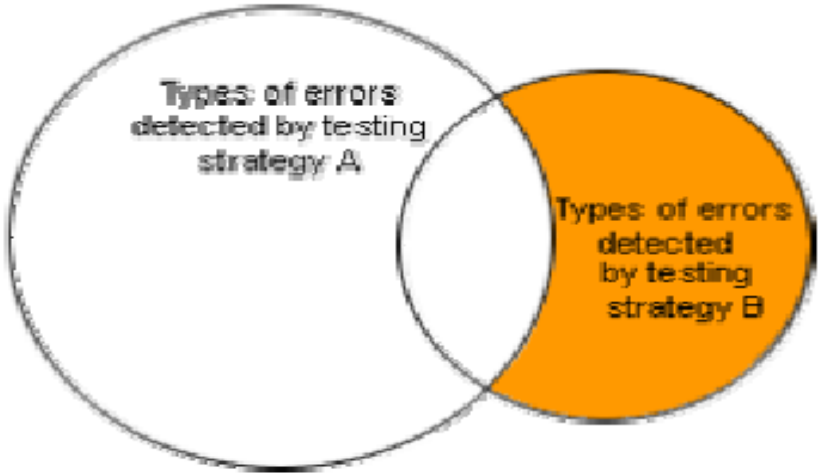
One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The concepts of stronger and complementary testing are schematically illustrated in fig. 10.2.

Here A **subsumes** B

Subsumes



A is a stronger testing strategy than B



A and B are complementary testing strategies

- **Statement coverage:** The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values.

- **Branch coverage:** In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program's control flow graph is traversed at least once.
- **Condition coverage:** In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values.

- **Path coverage:** The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

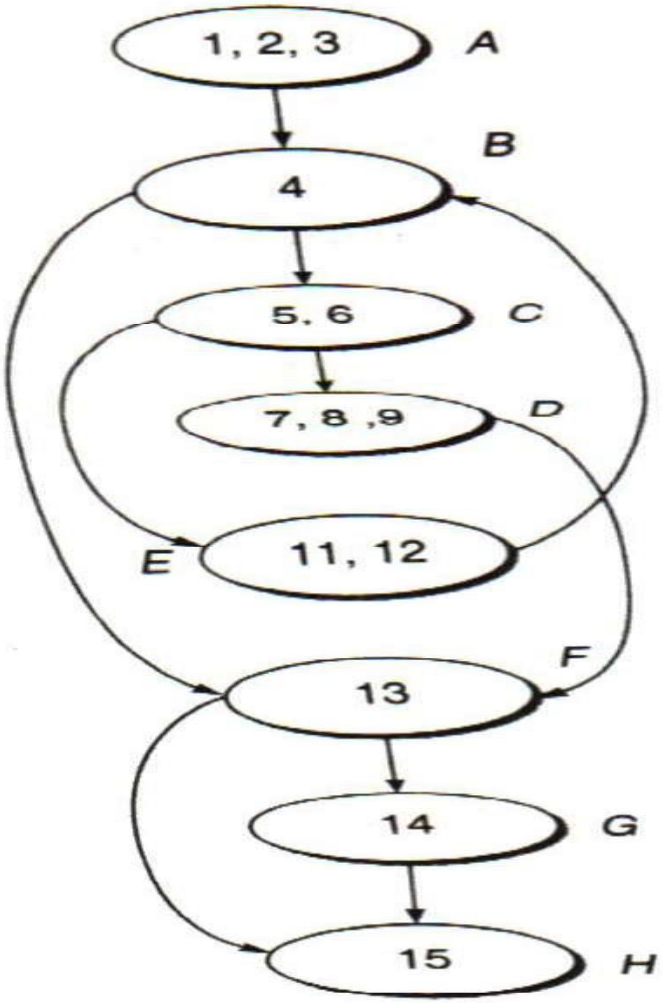
- **Control Flow Graph (CFG):** A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first.

Example

Consider the following program segment:

```
main()
{
      int number, index;
1.    printf("Enter a number");
2.    scanf("%d, &number);
3.    index = 2;
4.    while(index <= number - 1)
5.    {
6.          if (number % index == 0)
7.          {
8.                printf("Not a prime number");
9.                break;
10.         }
11.         index++;
12.         }

13.       if(index == number)
14.             printf("Prime number");
15. } //end main
```


1, 2, 3
A
B
4
5, 6
C
7, 8 ,9
D
E
11, 12
F
13
14
G
15
H

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-F-H

(ii) A-B-F-G-H

(iii) A-B-C-E-B-F-G-H

(iv) A-B-C-D-F-H

(d) Test case design from the list of independent paths

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Decision Table based Testing

Test Adequacy Measurement and Enhancement:

Control and Data flow

What is adequacy?

- Given a program P written to meet a set of functional requirements R = {R1, R2, …, Rn}.

We now ask:
Has P been tested thoroughly? or: Is T adequate?

In software testing, the terms "thorough" and "adequate" have the same meaning.

Measurement of adequacy

- Adequacy is measured for a given test set and a given criterion.

- A test set is considered adequate wrt criterion C when it satisfies C.

Example

Program sumProduct must meet the following requirements:

R1 Input two integers, x and y, from standard input.

R2.1 Print to standard output the sum of x and y if $x<y$.

R2.2 Print to standard output the product of x and y if $x \geq y$.

Example (contd.)

Suppose now that the test adequacy criterion C is specified as:

C : A test set T for program (P, R) is considered adequate if, for each r in R there is a test case in T that tests the correctness of P with respect to r.

T={t: <x=2, y=3> is *inadequate* with respect to C for program sumProduct.

The lone test case t in T tests R1 and R2.1, but not R2.2.

Black-box and white-box criteria

Given an adequacy criterion C, we derive a finite set Ce known as the coverage domain.

A criterion C is a white-box test adequacy criterion if the corresponding Ce depends solely on the program P under test.

A criterion C is a black-box test adequacy criterion if the corresponding Ce depends solely on the requirements R for the program P under test.

Coverage

Measuring adequacy of T:

T covers Ce if, for each e' in Ce, there is a test case in T that tests e'. T is adequate wrt C if it covers all elements of Ce.

T is inadequate with respect to C if it covers k<n elements of Ce.

k/n is the coverage of T wrt C.

Example

Consider criterion:
"A test T for (P, R) is adequate if , for each requirement r in R, there is at least one test case in T that tests the correctness of P with respect to r."

The coverage domain is Ce={R1, R2.1, R2.2}.
T covers R1 and R2.1 but not R2.2.
Hence, T is inadequate with respect to C.
The coverage of T wrt C is 2/3=0.66.

Another Example

Consider the following criterion:
"A test T for program (P, R) is considered adequate if each path in P is traversed at least once."

Assume that P has exactly two paths, p1 and p2, corresponding to condition $x<y$ and condition $x \geq y$, respectively.

For the given adequacy criterion C we obtain the coverage domain Ce to be the set {p1, p2}.

Another Example (contd.)

To measure the adequacy of T of sumProduct against C, we execute P against each test case in T .

As T contains only one test for which x<y, only path p1 is executed.

Thus, the coverage of T wrt C is 0.5. T is not adequate with respect to C.

We can also say that p1 is tested and p2 is not tested.

Code-based coverage domain

In the previous example, we assumed that P contains two paths.

When the coverage domain must contain elements from the code, these elements must be derived by analyzing the code.

Errors in the program and incomplete/incorrect requirements can cause the coverage domain to be different from the expected.

Example

sumProduct1

```
1    begin
2      int x, y;
3      input (x, y);
4      sum=x+y;
5      output (sum);
6    end
```

This program is incorrect as per the requirements of sumProduct.

- There is only one path, p1, which traverses all the statements.
- Using the path-based coverage criterion C, Ce={ p1}.
- T={t: <x=2, y=3>}is adequate w.r.t. C but does not reveal the error.

Example (contd.)

sumProduct2

```
1    begin
2      int x, y;
3      input (x, y);
4      if(x<y)
5      then
6        output(x+y);
7      else
8        output(x*y);
9    end
```

This program is correct as per the requirements of sumProduct.
It has two paths, p1 and p2.

Ce={ p1, p2}.
T={t: <x=2, y=3>} is inadequate w.r.t. the path-based coverage criterion C.

Lesson

An adequate test set might not reveal even the most obvious error in a program.

This does not diminish in any way the need for the measurement of test adequacy, as increasing coverage might reveal an error!

Debugging, Integration and System Testing

Need for debugging

- Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

Debugging guidelines

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistakes that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

Integration testing

- The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module.
- During integration testing, different modules of a system are integrated in a planned manner using an integration plan.
- The integration plan specifies the steps and the order in which modules are combined to realize the full system.
- After each integration step, the partially integrated system is tested. An important factor that guides the integration plan is the module dependency graph.
- The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart the integration plan can be developed.

Integration test approaches

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

- Big bang approach
- Top-down approach
- Bottom-up approach
- Mixed-approach

Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

Bottom-Up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

Top-Down Integration Testing

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

Test Stub and Test Driver

Brief introduction about Test stub and Test Driver

Role of these in integration and bottom up integration testing

System testing

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

- **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.
- **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the SRS document. Broadly, these tests can be classified into functionality and performance tests. The functionality tests test the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance tests test the conformance of the system with the nonfunctional requirements of the system.

Performance testing

Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document. There are several types of performance testing. Among of them nine types are discussed below. The types of performance testing to be carried out on a system depend on the different non-functional requirements of the system documented in the SRS document. All performance tests can be considered as black-box tests.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing
- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

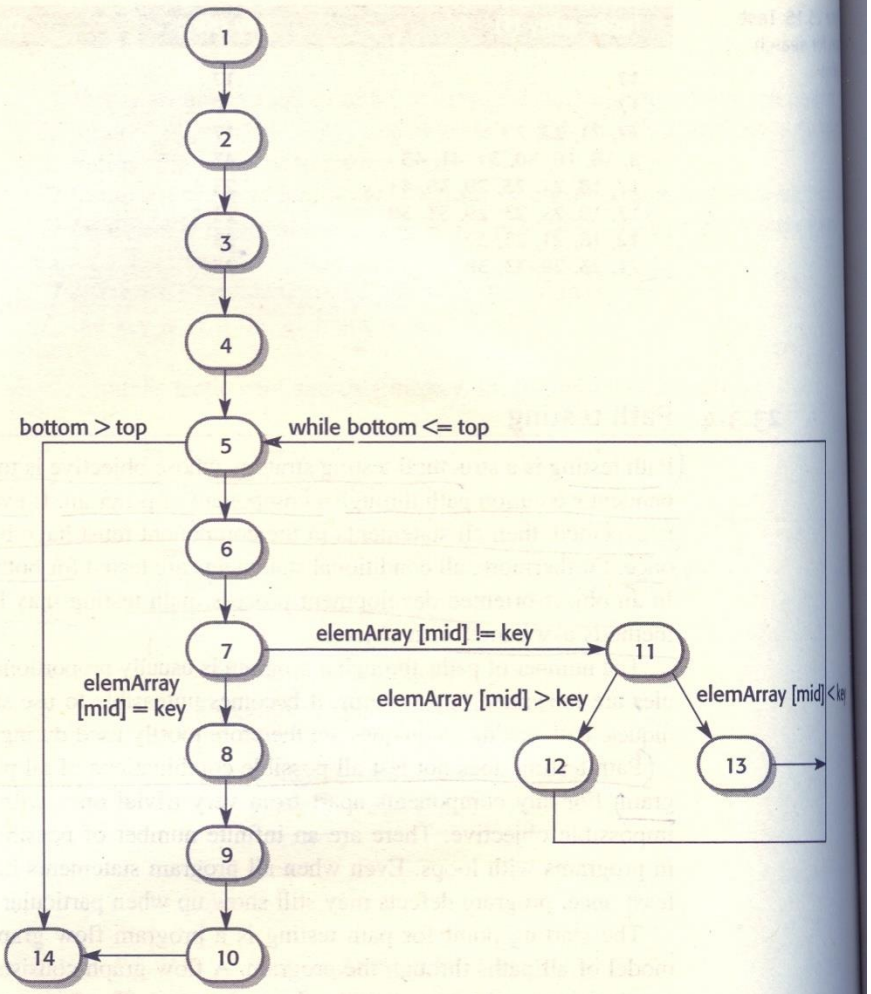
```
class BinSearch {

// This is an encapsulation of a binary search function that takes an array of
// ordered objects and a key and returns an object with 2 attributes namely
// index - the value of the array index
// found - a boolean indicating whether or not the key is in the array
// An object is returned because it is not possible in Java to pass basic types
by
// reference to a function and so return two values
// the key is -1 if the element is not found

      public static void search ( int key, int [] elemArray, Result r )
      {
1.        int bottom = 0 ;
2.        int top = elemArray.length - 1 ;
          int mid ;
3.        r.found = false ;
4.        r.index = -1 ;
5.        while ( bottom <= top ) && r.found == false)
          {
6.            mid = (top + bottom) / 2 ;
7.            if (elemArray [mid] == key)
              {
8.                r.index = mid ;
9.                r.found = true ;
10.                 return ;
               } // if part
               else
               {
11.                if (elemArray [mid] < key)
12.                    bottom = mid + 1 ;
                   else
13.                    top = mid - 1 ;
               }
          } //while loop
14.   } // search
      } //BinSearch
```

National Institute of Technology
Durgapur, India

Figure 23.16 Flow graph for a binary search routine



Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??