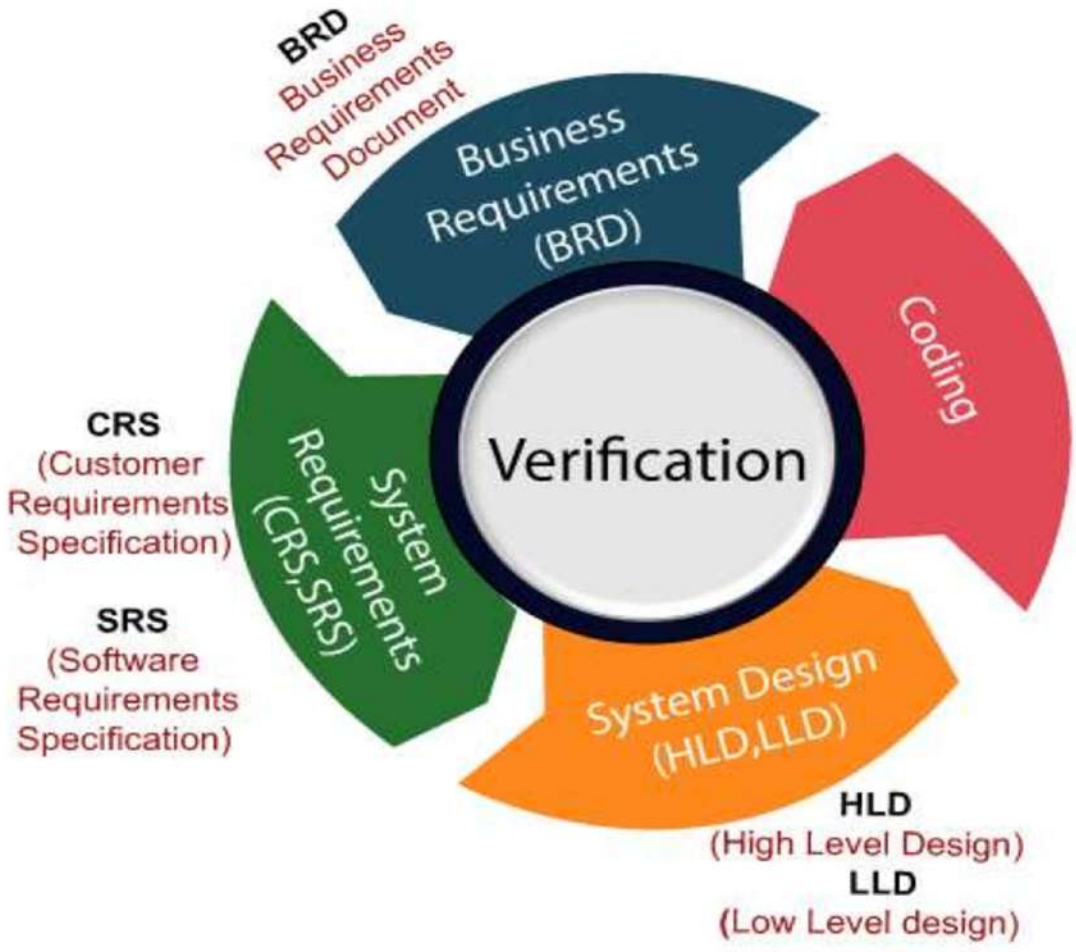
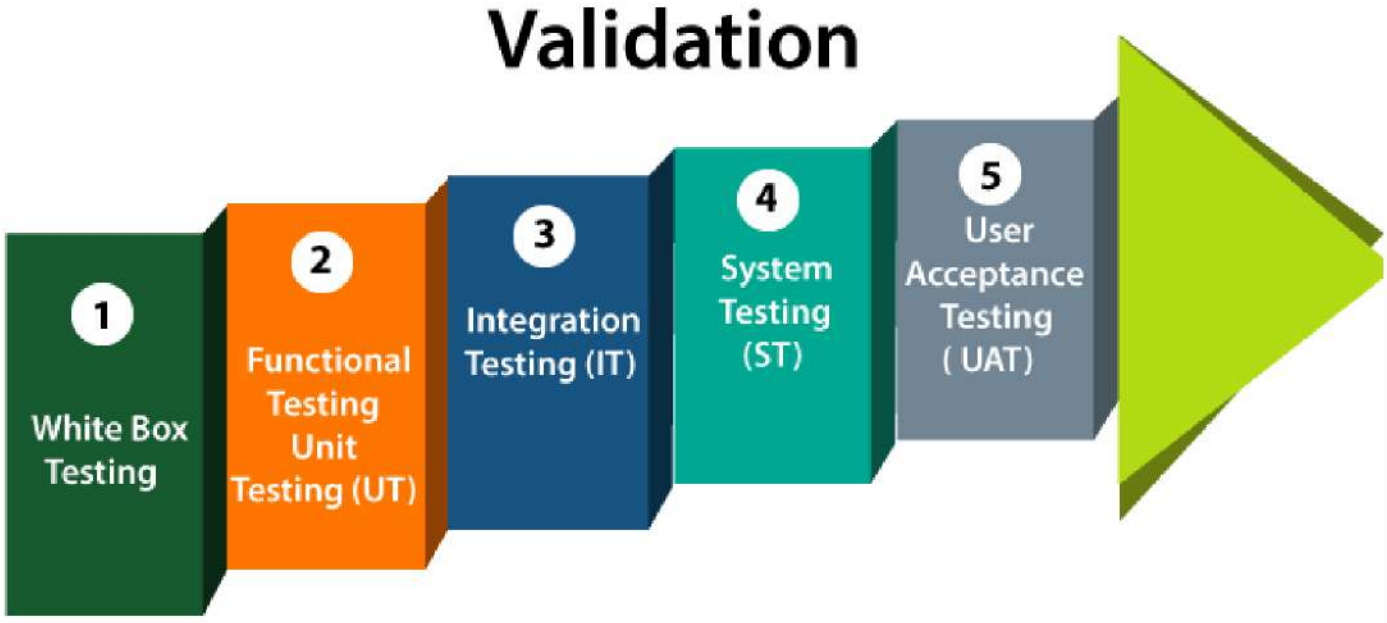
TESTING STRATEGIES

- **General characteristics of Software Testing**
- –Formal Technical Reviews
- –Testing begins at the component level and works "outward" towards the integration of the computer system
- –Diff. testing techniques appropriate at diff. points in time
- **•Verification and Validation**
- –Verification refers to the set of activities that ensure that software correctly implements a specific function
- –Validation refers to the set of activities that ensure that the software built is traceable to customer requirements
- –Boehm states this in another way:
- **Verification: are we building the right product?**
- **Validation : are we building the product right?**

- Verification



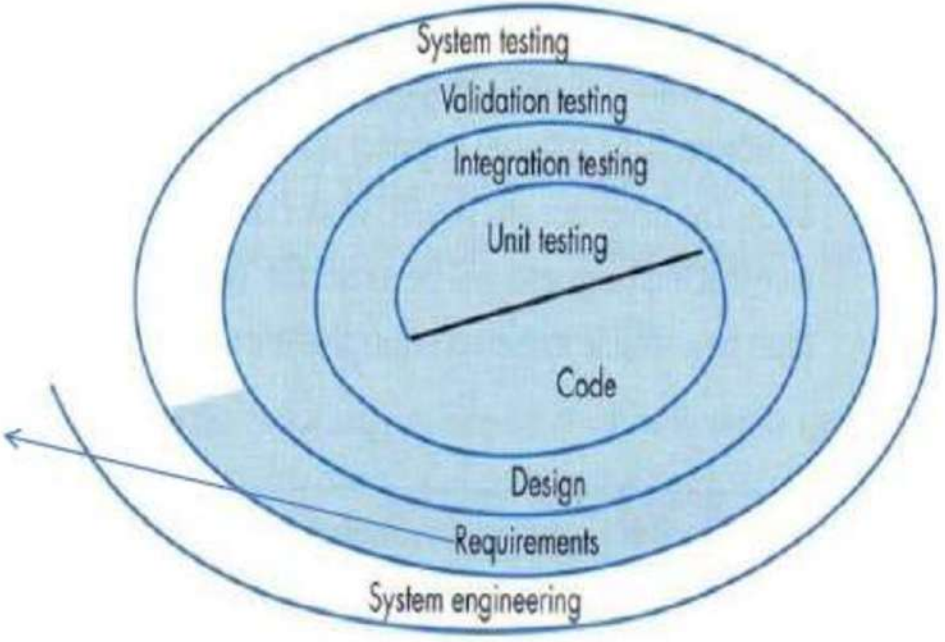
- Validation


Validation
1
White Box Testing
2
Functional Testing
Unit Testing (UT)
3
Integration Testing (IT)
4
System Testing (ST)
5
User Acceptance Testing (UAT)

- A Software Testing strategy for Conventional Software Architectures

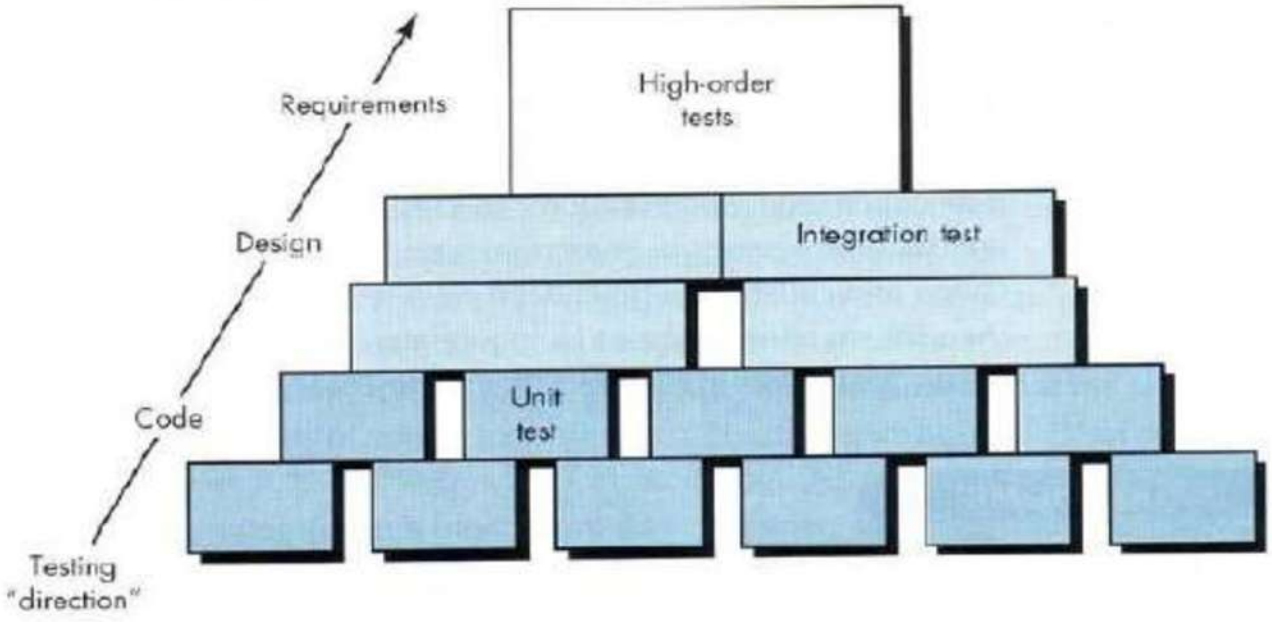
FIGURE 13.1

Testing strategy



Information domain, function, behavior, performance, constraints, and validation criteria for software are established

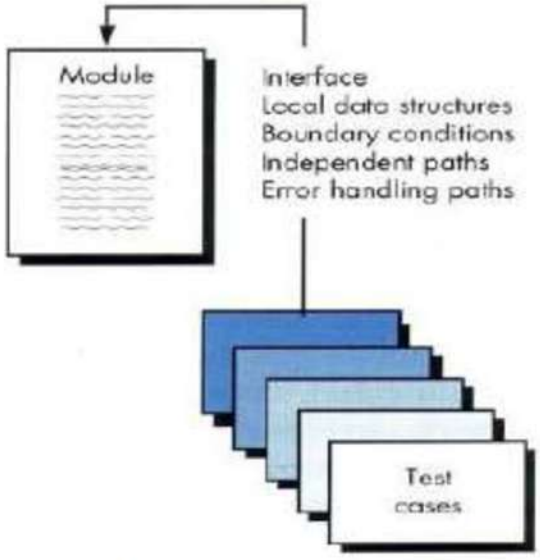
FIGURE 13.2
Software testing steps



- A Software Testing strategy for Object-oriented Software Architectures
- Criteria for Completion of Testing
  - When are we done testing? No definitive answer

- Test strategies for Conventional Software
- –It takes an incremental view of testing, beginning with
    - •The testing of individual program units
    - •Moving to tests designed to facilitate the integration of the units &
    - •Culminated with tests that exercise the constructed system
- –Unit Testing
    - •Important control paths are tested to uncover errors within the boundary of the module
    - •Focuses on the internal processing logic and data structures within the boundaries of a component
    - •Selective Testing of execution paths is an essential task during the unit test

FIGURE 13.3
Unit test



Unit Test Considerations

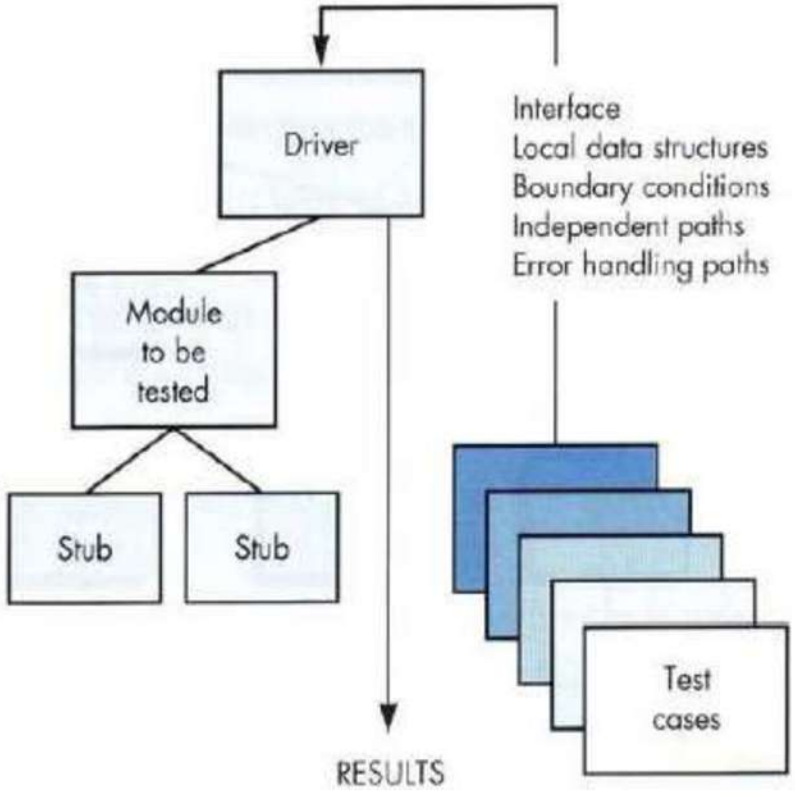
- Test cases designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow
- More common errors are
  - Incorrect arithmetic precedence
  - Mixed mode operations
  - Incorrect initialization
  - Precision inaccuracy
  - Incorrect symbolic representation of an expression

- Test cases
- Comparison of different data types
- –Incorrect logical operators or precedence
- –Expectation of equality when precision error makes equality unlikely
- –Incorrect comparison of variables
- –Improper or nonexistent loop termination
- –Failure to exit when divergent iteration is encountered
- –Improperly modified loop variables

- Unit test procedures
  - –Drivers and Stubs must be developed for each unit test
  - –Driver: a "main program" that accepts test case data, passes such data to the component, & prints relevant results
  - –Stubs: a "dummy subprogram" serve to replace modules that are subordinate to the component to be tested

FIGURE 13.4

Unit test environment



Drivers and Stubs represent overhead

Unit Testing is simplified when a component with high cohesion is designed

- Integration Testing
  - A systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing
  - Non incremental integration vs Incremental integration
  - **Top-down Integration**

Breadth-first integration

Depth-first integration

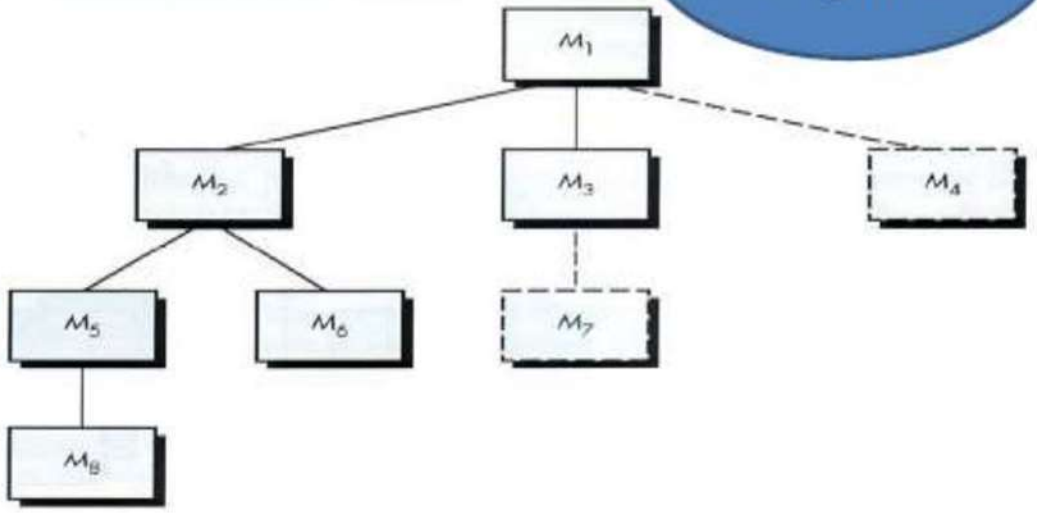


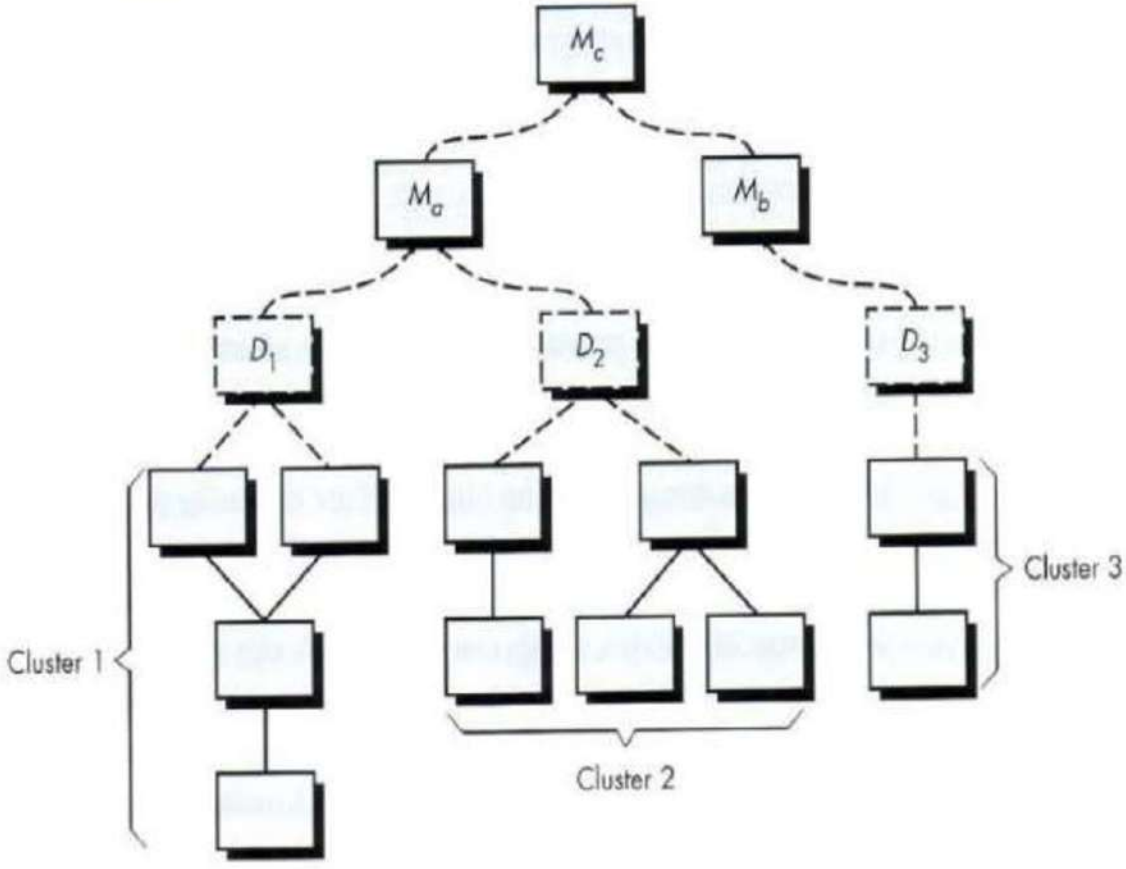
FIGURE 13.5
Top-down integration

- Integration process performed in a series of 5 steps:
  - Main control module used as a test driver & stubs substituted for all components directly subordinate to the main control module
  - Stubs replaced one at a time with actual components
  - Tests conducted as each component is integrated
  - On completion of each set of tests, another stub is replaced with real component
  - Regression testing conducted to ensure that new errors not been introduced.
- Top-down Integration relatively uncomplicated, but in practice logistical problems arise
- Most common problem: when processing at low levels in the hierarchy is required to test upper levels

- Tester has 3 choices
  - Delay many tests until stubs are replaced with actual modules,
  - Develop stubs that perform limited functions that simulate the actual module (or),
  - Integrate the software from bottom of the hierarchy upward
- Bottom-up Integration
  - Begins construction and testing with atomic modules
  - No need for stubs
- Each time a new module is added as part of integration testing, software changes
  - New data flow paths established,
  - New i/o may occur,
  - New control logic invoked
- Due to these changes, problems persist: hence Regression Testing has to be done

FIGURE 13.6

Bottom-up integration



- Regression Testing
  - is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
  - is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be conducted manually by re-executing a subset of all test cases or
  - Using automated capture/playback tools
  - Regression test suite contains 3 different classes of test cases:
    - A representative sample of tests that will exercise all software functions
    - Additional tests that focus on s/w functions that are likely to be affected by the change
    - Tests that focus on s/w components that have been changed

- Smoke Testing
  - Designed as a pacing mechanism for time-critical projects, allowing the software team to assess its projects on a frequent basis
  - It should exercise the entire system from end to end
  - Provides a no. of benefits when it is applied on complex, time-critical software engineering projects:
    - Integration risk is minimized
    - Quality of the end product is improved
    - Error diagnosis and corrections are simplified
    - Progress is easier to access

Smoke test is done to make sure that the critical functionalities of the program are working fine, whereas
Sanity testing is done to check that newly added functionalities, bugs, etc., have been fixed.

Smoke Testing has a goal to verify "stability" whereas Sanity Testing has a goal to verify "rationality".

Smoke testing is a subset of acceptance testing whereas Sanity testing is a subset of Regression Testing.

- As integration testing is conducted, tester should identify critical modules
- A critical module has one or more of the following characteristics:
  - Addresses several software requirements
  - Has a high level of control
  - Is complex or error-prone or
  - Has definitive performance requirements
- Critical modules should be tested as early as possible
- In addition, regression testing should focus on critical module functions
- Integration Test Documentation: Test SpecificationM

- Integration Testing in the OO Context
- –2 different strategies:
- •Thread-based Testing
- –Integrates the set of classes required to respond to one input or event for the system
- –Each thread is integrated and tested individually
- –Regression testing is applied to ensure that no side-effects occur
- •Use-based Testing
- –Testing those independent classes that use very few server classes
- –After that dependent classes that uses independent classes are then tested
- •Cluster Testing

- Validation Testing
- –Begins at the culmination of Integration Testing, when
- •Individual components are exercised,
- •Software is completely assembled as a package, &
- •Interfacing errors are uncovered and corrected
- –Focus is on user-visible actions and user-recognizable output from the system
- –Validation succeeds when software functions in a manner that can be reasonably acceptable by the customer
- –How to determine the "reasonable expectations"?
- –Validation Criteria
- –Configuration Review
- –Alpha and Beta Testing

- System Testing
- –A series of different tests whose primary purpose is to fully exercise the computer-based system

–Types of system testing

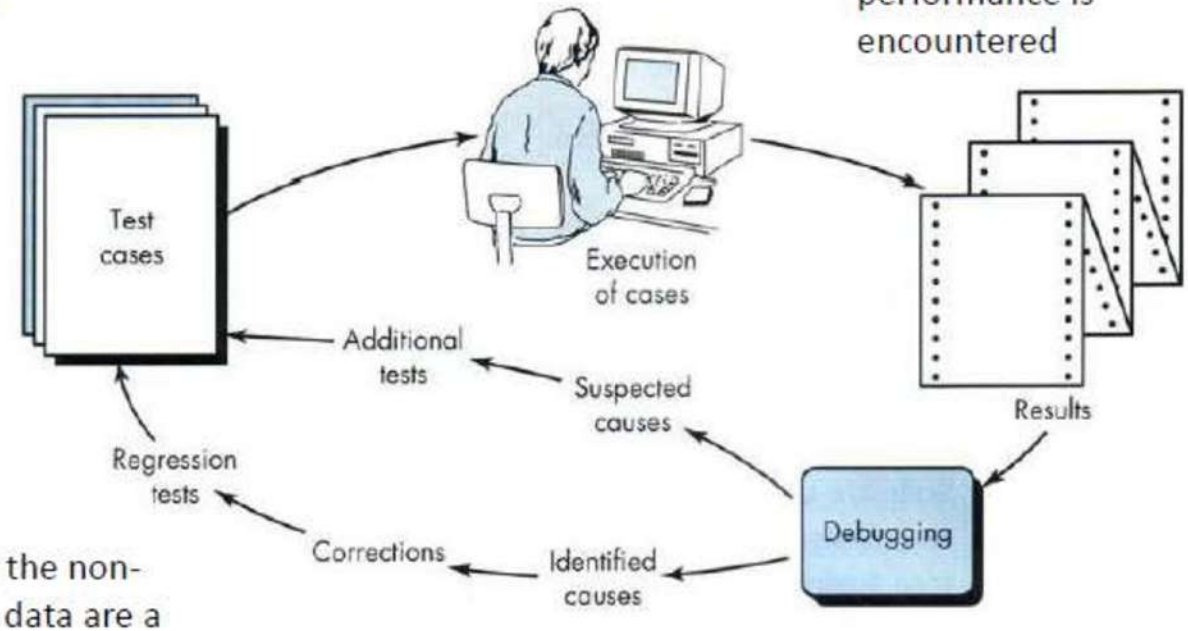
- **•Recovery testing**
- –System must be fault tolerant
- –In most cases, a system failure corrected within a specific period of time or severe economic damage will occur
- –Forces the system to fail in a variety of ways and verifies that recovery is properly performed
- –For automatic recovery: reinitialization, checkpointingmechanisms, data recovery, and restart evaluated for correctness
- –For manual recovery: MTTR evaluated to determine whether it is within acceptable limits

- **Security Testing**
- –Any system are subjected to harmful penetration
- –Penetration spans a broad range of activities:
- »Hackers penetration to systems for stealing confidential data
- »Annoyed employees for revenge
- »For illicit personal gain
- –verifies that protection mechanisms built into the system will protect it from improper penetration
- –Testers penetrate the system through various means
- –System designer to make penetration cost more than the value of the information that will be obtained
- **Stress Testing**
- –Designed to confront programs with abnormal situations

- **Performance Testing**
- –Is designed to test the run-time performance of software within the context of an integrated system
- –Occurs throughout all the steps in the testing process
- –Often coupled with stress testing and usually require both hardware and software instrumentation
- –Execution intervals
- –Log events
- **•Debugging**
- –Is not testing, but occurs as a consequence of successful testing
- –When a test case uncovers an error, debugging is an action that results in the removal of the error

Results are assessed and a lack of correspondence between expected and actual performance is encountered

FIGURE 13.7
The debugging process



In many cases, the non-corresponding data are a **symptom** of an underlying **cause** as yet hidden

Debugging attempts to match **symptom** with **cause**, thereby leading to error correction

- Debugging always have one of 2 outcomes:
- •The cause will be found and corrected (OR)
- •The cause will not be found
- –In the latter case, the person performing debugging may suspect a cause,

•design one or more test cases to help validate the suspicion, and Work toward error correction in an iterative fashion

TESTING TACTICS

- Testing Tactics –Test Case Design
- •How to write an effective test case?
- •Test case design
- •Characteristics of "Testability"
- –Software Testability is simply how easily a computer program can be tested
- –Must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum effort
- –Operability: "the better it works, the more efficiently it can be tested"
- –Observability: "What you see is what you test"
- –Controllability: "the better we can control the software, the more the testing can be automated and optimized"
- –Decomposability:
- –Simplicity:
- –Stability:
- –Understandability:

- Test characteristics
- –A good test has a high probability of finding an error
- –A good test is not redundant
- –A good test should be "best of breed"
- –A good test should be neither too simple nor too complex
- •Black-box Testing and White-Box Testing
- –BB Testing
- The Black Box Test is a test that only considers the external behavior of the system; the internal workings of the software is not taken into account. It is a functional test of the software.
- --WB testing
- The White Box Test is a method used to test a software taking into consideration its internal functioning. It is a structural test of the software.

- **Black-box test design techniques-**
- Decision table testing
- Equivalence partitioning
- Boundary Value Analysis

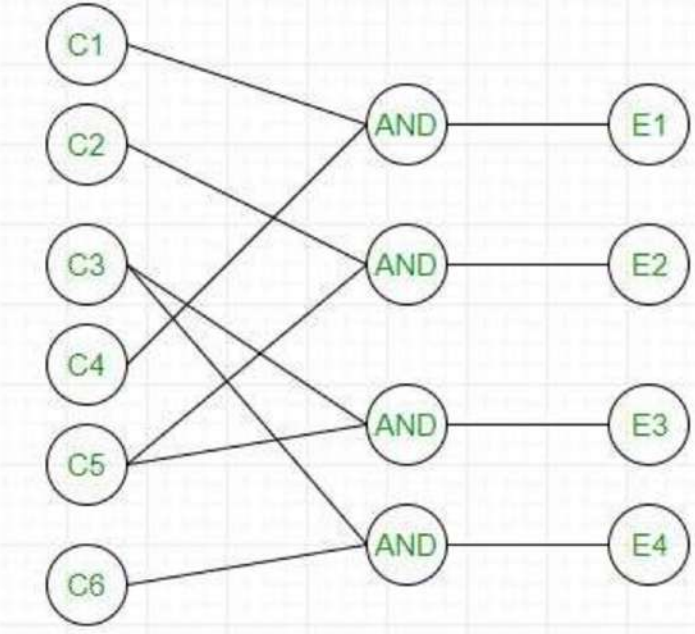
Types of Black Box Testing:

- Functional Testing
- Non-functional testing
- Regression Testing

- **Equivalence partitioning** – It is often seen that many types of inputs work similarly so instead of giving all of them separately we can group them and test only one input of each group. The idea is to partition the input domain of the system into several equivalence classes such that each member of the class works similarly, i.e., if a test case in one class results in some error, other members of the class would also result in the same error.
- The technique involves two steps:
- **Identification of equivalence class** – Partition any input domain into a minimum of two sets: **valid values** and **invalid values**. For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.
- **Generating test cases** – (i) To each valid and invalid class of input assign a unique identification number. (ii) Write a test case covering all valid and invalid test cases considering that no two invalid inputs mask each other. To calculate the square root of a number, the equivalence classes will be **(a) Valid inputs:**
  - The whole number which is a perfect square- output will be an integer.
  - The whole number which is not a perfect square- output will be a decimal number.
  - Positive decimals
  - Negative numbers(integer or decimal).
  - Characters other than numbers like “a”,”!”,”;”, etc.

- **Boundary value analysis** – Boundaries are very good places for errors to occur. Hence if test cases are designed for boundary values of the input domain then the efficiency of testing improves and the probability of finding errors also increases. For example – If the valid range is 10 to 100 then test for 10,100 also apart from valid and invalid inputs.
- **Cause effect Graphing** – This technique establishes a relationship between logical input called causes with corresponding actions called the effect. The causes and effects are represented using Boolean graphs. The following steps are followed:
- Identify inputs (causes) and outputs (effect).
- Develop a cause-effect graph.
- Transform the graph into a decision table.
- Convert decision table rules to test cases.

- For example, in the following cause-effect graph:
- It can be converted into a decision table like:



		1	2	3	4
CAUSES	C1	1	0	0	0
	C2	0	1	0	0
	C3	0	0	1	1
	C4	1	0	0	0
	C5	0	1	1	0
	C6	0	0	0	1
EFFECTS	E1	X	-	-	-
	E2	-	X	-	-
	E3	-	-	X	-
	E4	-	-	-	X

- Each column corresponds to a rule which will become a test case for testing. So there will be 4 test cases.

- **Black Box Testing Type**
- The following are the several categories of black box testing:
- Functional Testing
- Regression Testing
- Nonfunctional Testing (NFT)
- **Functional Testing:** It determines the system's software functional requirements.

  Regression Testing: It ensures that the newly added code is compatible with the existing code. In other words, a new software update has no impact on the functionality of the software. This is carried out after a system maintenance operation and upgrades.
  Nonfunctional Testing: Nonfunctional testing is also known as NFT. This testing is not functional testing of software. It focuses on the software's performance, usability, and scalability.

- **Tools Used for Black Box Testing:**
- Appium
- Selenium
- Microsoft Coded UI
- Applitools
- HP QTP.

- **Features of black box testing:**
- **Independent testing:** Black box testing is performed by testers who are not involved in the development of the application, which helps to ensure that testing is unbiased and impartial.
- **Testing from a user's perspective:** Black box testing is conducted from the perspective of an end user, which helps to ensure that the application meets user requirements and is easy to use.
- **No knowledge of internal code:** Testers performing black box testing do not have access to the application's internal code, which allows them to focus on testing the application's external behavior and functionality.
- **Requirements-based testing:** Black box testing is typically based on the application's requirements, which helps to ensure that the application meets the required specifications.

- **Features of black box testing: -contd…**
- **Different testing techniques:** Black box testing can be performed using various testing techniques, such as functional testing, usability testing, acceptance testing, and regression testing.
- **Easy to automate:** Black box testing is easy to automate using various automation tools, which helps to reduce the overall testing time and effort.
- **Scalability:** Black box testing can be scaled up or down depending on the size and complexity of the application being tested.
- **Limited knowledge of application:** Testers performing black box testing have limited knowledge of the application being tested, which helps to ensure that testing is more representative of how the end users will interact with the application.

- **Advantages of Black Box Testing:**
- The tester does not need to have more functional knowledge or programming skills to implement the Black Box Testing.
- It is efficient for implementing the tests in the larger system.
- Tests are executed from the user's or client's point of view.
- Test cases are easily reproducible.
- It is used in finding the ambiguity and contradictions in the functional specifications.

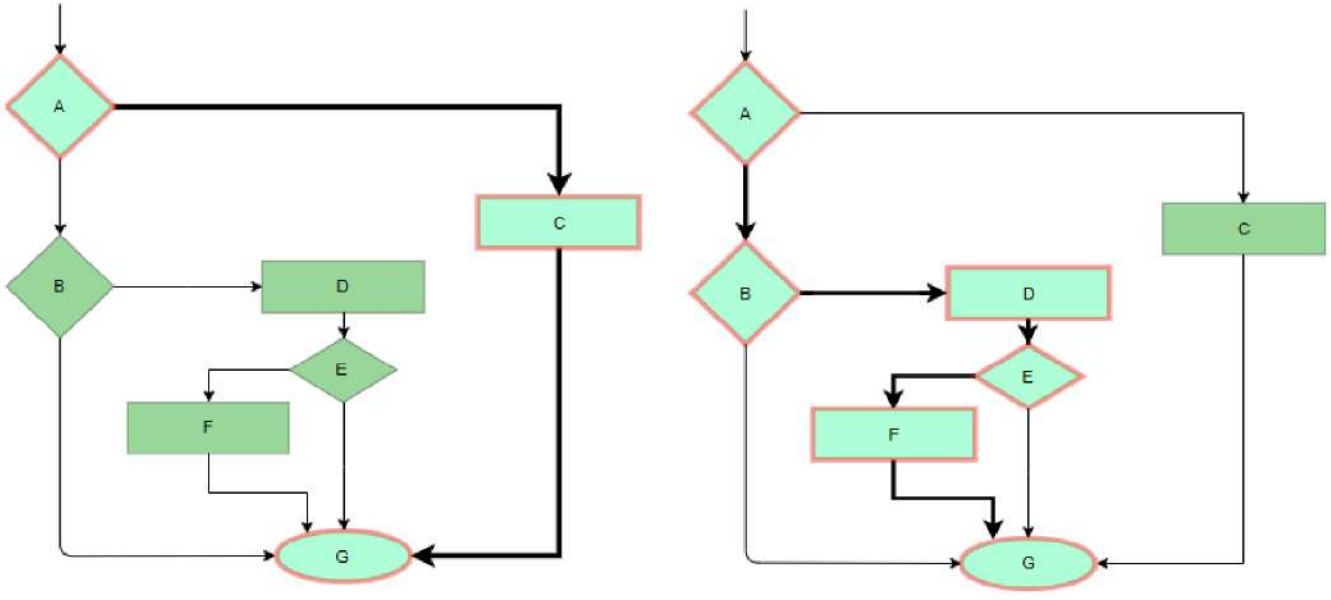
- **Disadvantages of Black Box Testing:**
- There is a possibility of repeating the same tests while implementing the testing process.
- Without clear functional specifications, test cases are difficult to implement.
- It is difficult to execute the test cases because of complex inputs at different stages of testing.
- Sometimes, the reason for the test failure cannot be detected.
- Some programs in the application are not tested.
- It does not reveal the errors in the control structure.
- Working with a large sample space of inputs can be exhaustive and consumes a lot of time

White-Box Testing

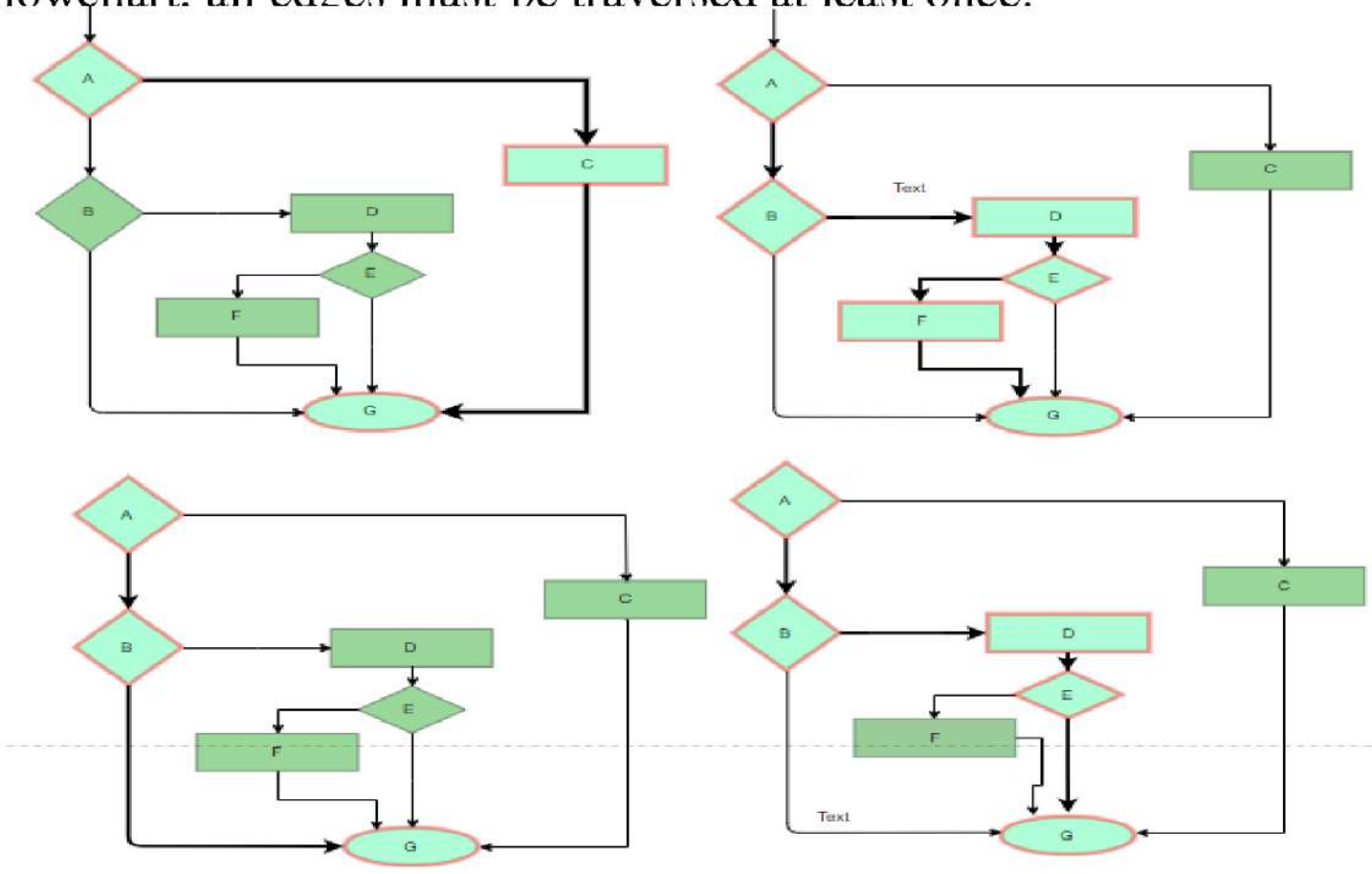
- –Also called as Glass-Box testing, is a test-case design philosophy that uses the control structure described as part of the component-level design to derive test cases
- •Guarantee that all independent paths within a module have been exercised at least once
- •Exercise all logical decisions on their true and false sides
- •Execute all loops at their boundaries and within their operational bounds
- •Exercise internal data structures to ensure their validity
- –Basic Path Testing
- •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

- **White-box test design techniques-**
- Control flow testing
- Data flow testing
- Branch testing
- **Types of White Box Testing:**
- Path Testing
- Loop Testing
- Condition testing

- **Statement coverage:** In this technique, the aim is to traverse all statements at least once. Hence, each line of code is tested. In the case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



- **Branch Coverage:** In this technique, test cases are designed so that each branch from all decision points is traversed at least once. In a flowchart, all edges must be traversed at least once.



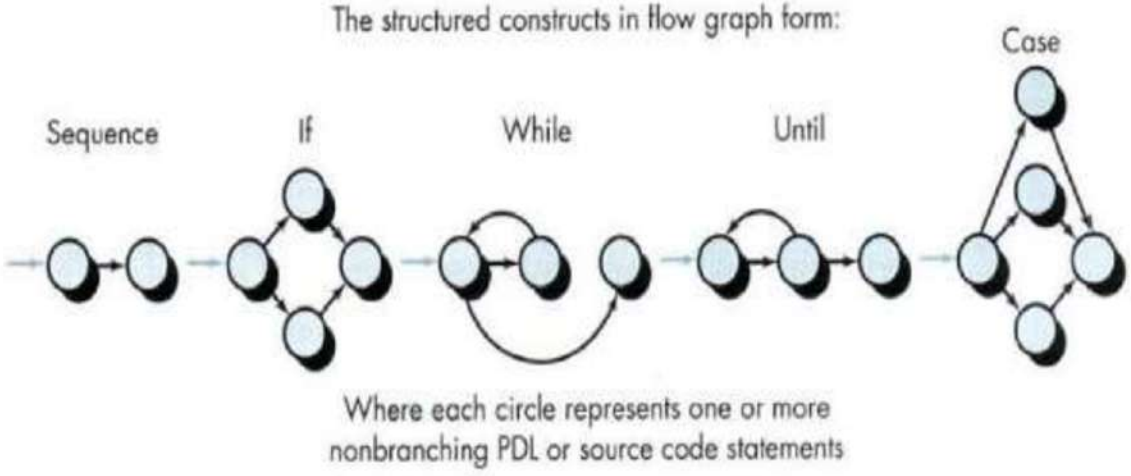
- **Condition Coverage:** In this technique, all individual conditions must be covered as shown in the following example:
  - READ X, Y
  - IF(X == 0 || Y == 0)
  - PRINT '0'
  - #TC1 – X = 0, Y = 55
  - #TC2 – X = 5, Y = 0
- **Multiple Condition Coverage:** In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let's consider the following example:
  - READ X, Y
  - IF(X == 0 || Y == 0)
  - PRINT '0'
  - #TC1: X = 0, Y = 0
  - #TC2: X = 0, Y = 5
  - #TC3: X = 55, Y = 0
  - #TC4: X = 55, Y = 5

- **Basis Path Testing:** In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.
- **Steps:**
  - Make the corresponding control flow graph
  - Calculate the cyclomatic complexity(A software metric that provides the quantitative measure of the logical complexity of a program)
  - Find the independent paths
  - Design test cases corresponding to each independent path
  - V(G) = P + 1, where P is the number of predicate nodes in the flow graph
  - V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes
  - V(G) = Number of non-overlapping regions in the graph

- Test cases derived to exercise the basis set are guaranteed to execute every statement at least one time during testing

- Flow Graph notation
  - Flow Graph depicts logical control flow

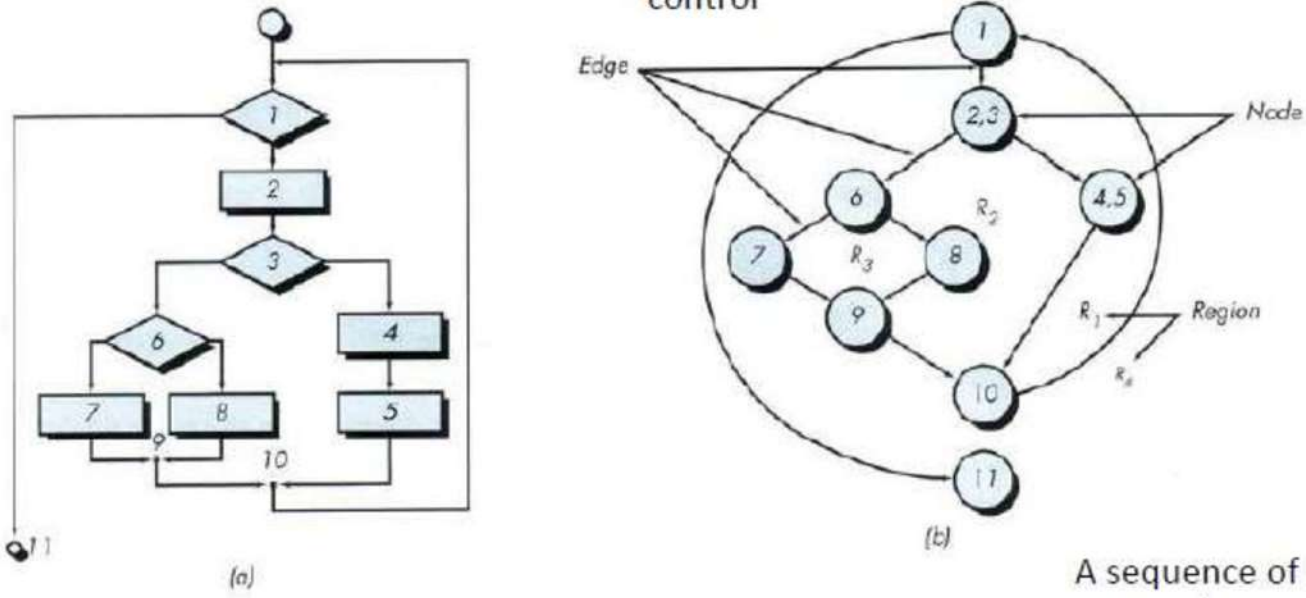
FIGURE 14.1
Flow graph notation



Each circle, called a Flowgraph node represents one or more procedural statements

Or links represent flow of control

FIGURE 14.2 (a) Flowchart and (b) flow graph

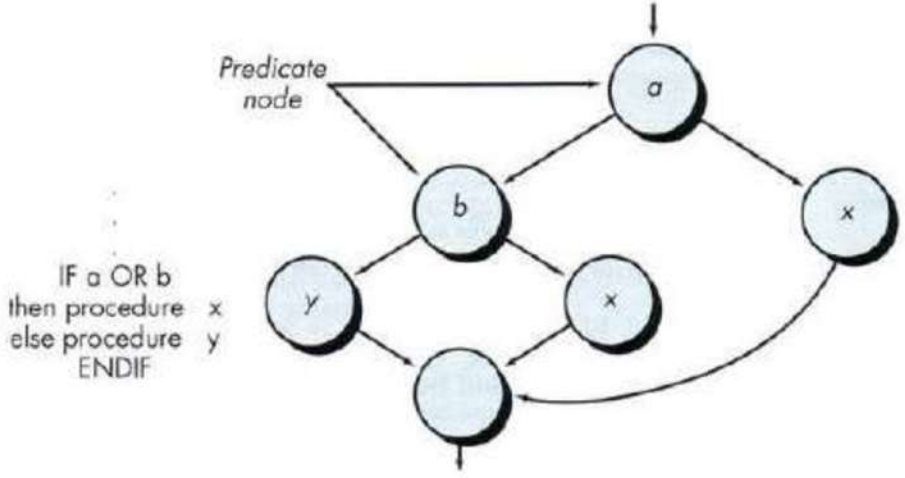


Maps the flowchart into a corresponding flowgraph

A sequence of process boxes and a decision diamond can map into a single node

- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated
- Each node that contains a condition is called a Predicate Node & is characterized by 2 or more edges deriving from it

FIGURE 14.3
Compound logic

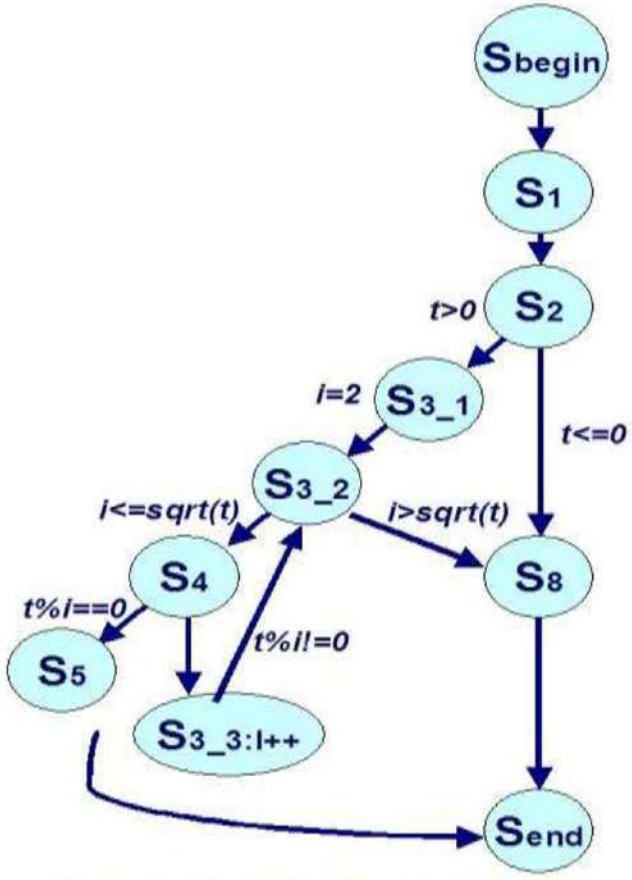


- Independent Program Paths
  - Any path through the program that introduces at least one new set of processing statements or a new condition
  - An independent path must move along at least one edge that has not been traversed before the path is defined
  - Example: from fig. 14.2 (b)
    - Path 1: 1-11
    - Path 2: 1-2-3-4-5-10-11
    - Path 3: 1-2-3-6-8-9-10-1-11
    - Path 4: 1-2-3-6-7-9-10-1-11

    Each new path introduces a new edge
  - Path 1,2,3, and 4 constitute a basis set for the flow graph
  - This basis set is not unique

- 1.The no. of regions corresponds to the cyclomaticcomplexity –V(G)
- 2.V(G) for a flow graph, G, is defined as
- V(G) = E –N + 2
- E –no. of flow graph edges
- N –no. of flow graph nodes
- 3.CyclomaticComplexity, V(G), for a flow graph, G, is also defined as, V(G) = P + 1
- P –no. of predicate nodes contained in the flow graph, G.
- 4.From Fig.14.2b, the cyclomaticcomplexity can be computed using each of the algorithms
- flow graph has 4 regions
- V(G) = 11 edges –9 nodes + 2 = 4
- V(G) = 3 predicate nodes + 1 = 4

```
S1  Boolean Composite (integer t)
S2  if (t > 0) {
S3    for (i=2; i<=sqrt(t); i++) {
S4      if (t%i==0) {
S5        return true; }
S6    }
S7  }
S8  return false;
```



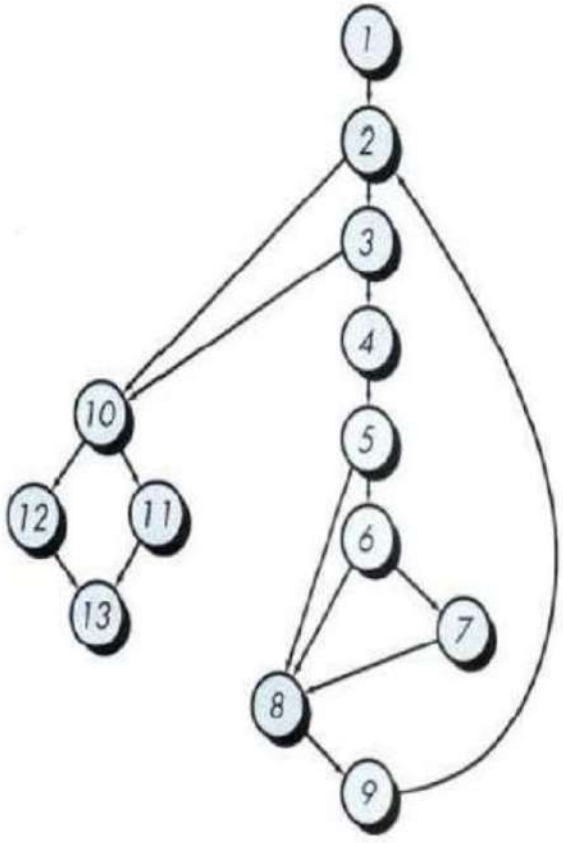
Pseudo Code to find out if an Integer is a composite Number

```
PROCEDURE average;

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

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

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



- **Data Flow Testing** is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.

It is concerned with:

- Statements where variables receive values,
- Statements where these values are used or referenced.
- To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S-

```
DEF(S) = {X | statement S contains the definition of X}
USE(S) = {X | statement S contains the use of X}
```

Variable occurrence can be one of the following three types

- **def** represents the definition of a variable. The variable on the left-hand side

of an assignment statement is the one getting defined .

- ***c-use*** represents computational use of a variable. Any statement (e.g., read, write, an assignment) that uses the value of variables for computational purposes

is said to be making c-use of the variables. In an assignment statement, all variables on the right-hand side have a c-use occurrence. In a read and a write statement, all variable occurrences are of this type.

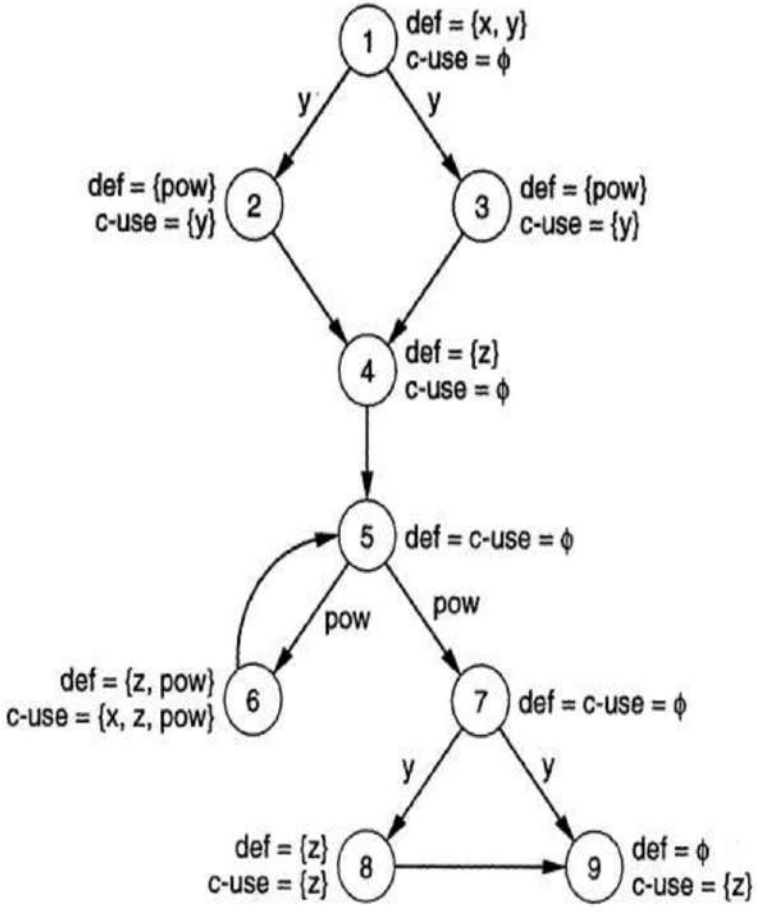
- ***p-use*** represents predicate use. These are all the occurrences of the variables in a predicate (i.e., variables whose values are used for computing the value of the predicate), which is used for transfer of control.

- If a statement is a loop or if condition then its DEF set is empty and USE set is based on the condition of statement s.
- Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program. Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables. These anomalies are:
- A variable is defined but not used or referenced,
- A variable is used but never defined,
- A variable is defined twice before it is used

- **Advantages of Data Flow Testing:**
  Data Flow Testing is used to find the following issues-
- To find a variable that is used but never defined,
- To find a variable that is defined but never used,
- To find a variable that is defined multiple times before it is use,
- Deallocating a variable before it is used.
- **Disadvantages of Data Flow Testing**
- Time consuming and costly process
- Requires knowledge of programming languages

```
1. scanf(x, y); if (y $<$ 0)
2.   pow = 0 - y;
3. else pow = y;
4. z = 1.0;
5. while (pow != 0)
6.   { z = z * x; pow = pow - 1; }
7. if (y $<$ 0)
8.   z = 1.0/z;
9. printf(z);
```

(node, var)	dcu	dpu
(1, x)	{6}	ϕ
(1, y)	{2, 3}	{(1,2), (1,3), (7, 8), (7, 9)}
(2, pow)	{6}	{(5, 6), (5, 7)}
(3, pow)	{6}	{(5, 6), (5, 7)}
(4, z)	{6, 8, 9}	ϕ
(6, z)	{6, 8, 9}	ϕ
(6, pow)	{6}	{(5, 6), (5, 7)}
(8, z)	{9}	ϕ



- **Loop Testing:** Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.
  - **Simple loops:** For simple loops of size n, test cases are designed that:
    - Skip the loop entirely
    - Only one pass through the loop
    - 2 passes
    - m passes, where m < n
    - n-1 ans n+1 passes
  - **Nested loops:** For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.
  - **Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

- **White Testing is Performed in 2 Steps:**
- 1. Tester should understand the code well
- 2. Tester should write some code for test cases and execute them
- **Tools required for White box Testing:**
- PyUnit
- Sqlmap
- Nmap
- Parasoft Jtest
- Nunit
- VeraUnit
- CppUnit

- **Features of white box testing:**
- **Code coverage analysis:** White box testing helps to analyze the code coverage of an application, which helps to identify the areas of the code that are not being tested.
- **Access to the source code:** White box testing requires access to the application's source code, which makes it possible to test individual functions, methods, and modules.
- **Knowledge of programming languages:** Testers performing white box testing must have knowledge of programming languages like Java, C++, Python, and PHP to understand the code structure and write tests.
- **Identifying logical errors:** White box testing helps to identify logical errors in the code, such as infinite loops or incorrect conditional statements.

- **Features of white box testing: contd…**
- **Integration testing:** White box testing is useful for integration testing, as it allows testers to verify that the different components of an application are working together as expected.
- **Unit testing:** White box testing is also used for unit testing, which involves testing individual units of code to ensure that they are working correctly.
- **Optimization of code:** White box testing can help to optimize the code by identifying any performance issues, redundant code, or other areas that can be improved.
- **Security testing:** White box testing can also be used for security testing, as it allows testers to identify any vulnerabilities in the application's code.

- **Advantages:**
- White box testing is thorough as the entire code and structures are tested.
- It results in the optimization of code removing errors and helps in removing extra lines of code.
- It can start at an earlier stage as it doesn't require any interface as in the case of black box testing.
- Easy to automate.
- White box testing can be easily started in Software Development Life Cycle.
- Easy Code Optimization.
- **Some of the advantages of white box testing include:**
- Testers can identify defects that cannot be detected through other testing techniques.
- Testers can create more comprehensive and effective test cases that cover all code paths.
- Testers can ensure that the code meets coding standards and is optimized for performance.

- **However, there are also some disadvantages to white box testing, such as:**
- Testers need to have programming knowledge and access to the source code to perform tests.

- Testers may focus too much on the internal workings of the software and may miss external issues.

- Testers may have a biased view of the software since they are familiar with its internal workings.
  Overall, white box testing is an important technique in software engineering, and it is useful for identifying defects and ensuring that software applications meet their requirements and specifications at the code level

- **Disadvantages:**
- It is very expensive.
- Redesigning code and rewriting code needs test cases to be written again.
- Testers are required to have in-depth knowledge of the code and programming language as opposed to black-box testing.
- Missing functionalities cannot be detected as the code that exists is tested.
- Very complex and at times not realistic.
- Much more chances of Errors in production.

PRODUCT METRICS

- PRODUCT METRICS

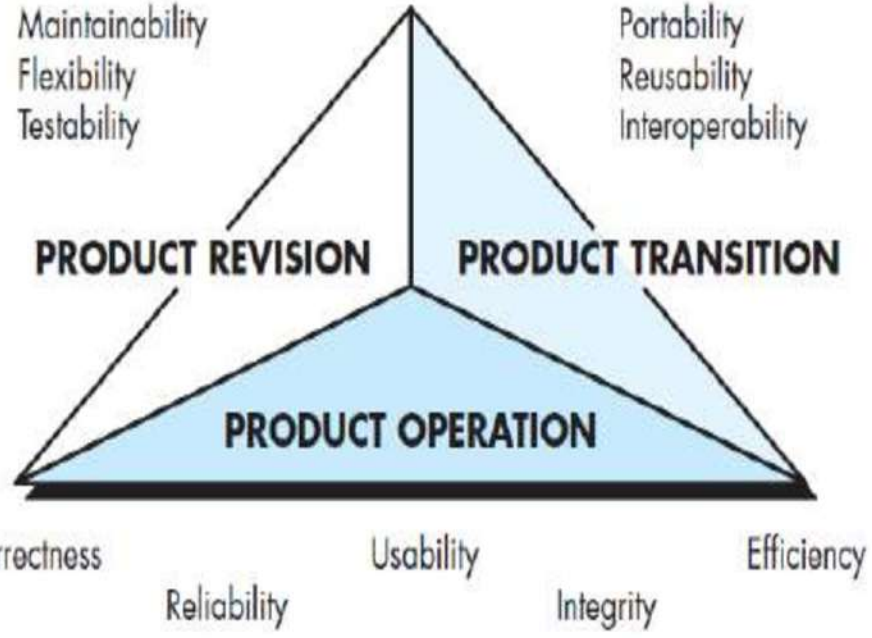
Direct measures are unusual in the software world

- •Software metrics are indirect
- •Measures used to assess the quality of the software product as it is being engineered
- •Software Quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- –Software requirements: foundation from which quality is measured; Lack of conformance to requirements is lack of quality
- –Specified standards: define a set of development criteria that guide the manager in which the software is engineered
- –If software conforms to its explicit requirements but fails to meet implicit requirements, quality is compromised

- Mccall's Quality factors
  - propose a useful categorization of factors that affect software quality.
  - focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments

FIGURE 19.1

McCall's software quality factors



- McCall and his colleagues provide the following descriptions:
- –*Correctness* -The extent to which a program satisfies its specification and fulfills the customer's mission objectives
- –*Reliability.*
- –*Efficiency.*
- –*Integrity* -Extent to which access to software or data by unauthorized persons can be controlled.
- –*Usability*
- –*Maintainability*
- –*Flexibility*
- –*Testability*
- –*Portability.*
- –*Reusability*
- –*Interoperability*
- •Difficult to develop direct measures of these quality factors

- ISO 9126 Quality Factors
- –Was developed to identify quality attributes for computer software
- –**Functionality:** degree to which the software satisfies stated needs as indicated by the following sub-attributes:
- •suitability, accuracy, interoperability, compliance, and security.
- –**Reliability:** amount of time that the software is available for use as indicated by the following sub-attributes
- •Maturity, fault tolerance, recoverability
- –**Usability:** degree to which the software is easy to use as indicated by the following sub-attributes
- •understandability, learnability, operability

- –**Efficiency:** degree to which the software makes optimal use of system resources as indicated by the following sub-attributes
- •Time behavior, resource behavior.
- –**Maintainability:**The ease with which repair may be made to the software as indicated by the following sub-attributes
- •analyzability, changeability, stability, testability.
- –**Portability:** The ease with which the software can be transposed from one environment to another as indicated by the following sub-attributes
- •adaptability, installability, conformance, replaceability
- •The transition to a Quantitative view
- –Quality is subjective, a more precise definition of software quality is needed as a way to derive quantitative measurements of software quality for objective analysis

- A framework for Product Metrics
- –Within the software engineering context,
- •***Measure***: provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- •***Measurement:*** the act of determining a measure.
- •***Metric***: "a quantitative measure of the degree to which a system, component, or process possesses a given attribute".
- –A software engineer collects measures and develops metrics so that indicators will be obtained
- –**Indicator:** a metric or combination of metrics that provides insight into the software process, a software project, or the product itself.
- –When a single data point has been collected (e.g., the number of errors uncovered within a single software component), a measure has been established

- Measurement occurs as the result of the collection of one or more data points
- (a number of component reviews and unit tests are investigated to collect measures of the number of errors for each).
- –Software metric relates the individual measures in some way (e.g., the average number of errors found per review or the average number of errors found per unit test)
- –What are the steps of an effective measurement process?
- •Formulation
- •Collection
- •Analysis
- •Interpretation
- •Feedback

- Software metrics are useful only if they are characterized effectively & validated so that their worth is proven:
- •A metric should have desirable mathematical properties
- •When a metric represents a software characteristic that increases when positive traits occur or decrease when undesirable traits are encountered, the value of metric should increase or decrease in the same manner

- Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions
- •The Product Metrics Landscape
- –**Metrics for the Analysis model**
- •Functionality delivered
- •System size
- •Specification quality
- –**Metrics for the Design model**
- •Architectural metrics
- •Component-level metrics
- •Interface design metrics
- •Specialized OO design metrics

- **–Metrics for Source code**

•Halstead metrics

•Complexity metrics

•Length metrics

-Metrics for testing

•Statement and Branch coverage metrics

•Defect-related metrics

•Testing Effectiveness

•In-process metrics

-Metrics for Maintenance

- SMI

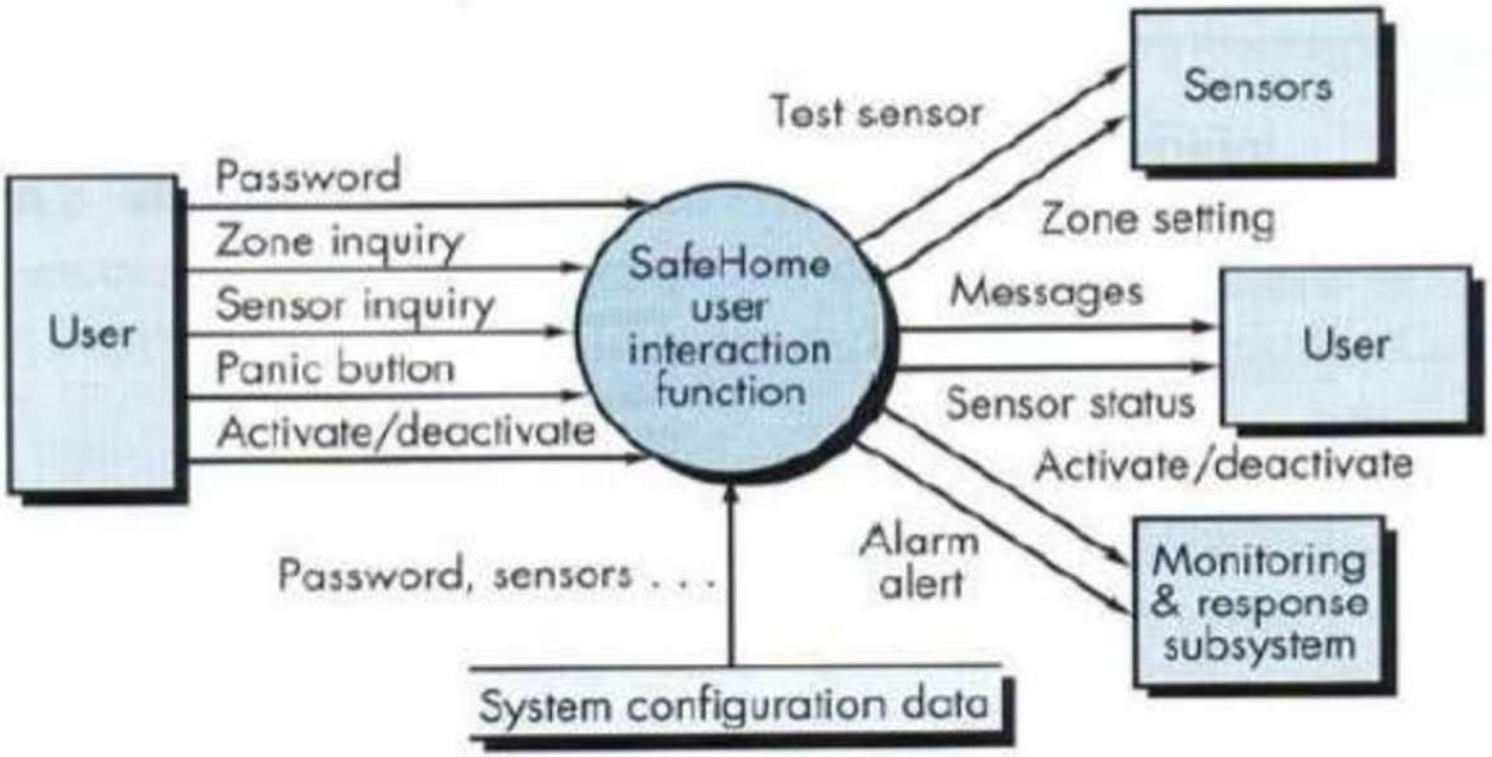
METRICS FOR REQUIREMENT MODEL

- Function-based Metrics
- •FP metric
- •Function points derived using an empirical relationship based on direct measures of software's information domain & assessments of software complexity
- •Information domain values:
- –No. of external inputs
- –No. of external outputs
- –No. of external queries
- –No. of internal logic files (ILF)
- –No. of external interface files (EIF)

- Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average or complex
- To compute FPs, the following relationship holds:
  - FP = count total x [0.65 + 0.01 x $\sum (F_i)$]
  - F_i (i=1 to 14) are value adjustment factors (VAF) based on responses to the following questions:

FIGURE 15.2
Computing function points

Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total							


Sensors
Test sensor
Zone setting
Password
Zone inquiry
Sensor inquiry
Panic button
Activate/deactivate
User
SafeHome user interaction function
Messages
User
Sensor status
Activate/deactivate
Alarm alert
Monitoring & response subsystem
Password, sensors . . .
System configuration data

- EI- 3
- EO –2
- EQ-2
- ILF-1
- EIF-3

FIGURE 15.2 Computing function points

Information Domain Value	Count		Weighting factor: Simple	Average	Complex	
External Inputs (EIs)		×	3	4	6	=
External Outputs (EOs)		×	4	5	7	=
External Inquiries (EQs)		×	3	4	6	=
Internal Logical Files (ILFs)		×	7	10	15	=
External Interface Files (EIFs)		×	5	7	10	=
Count total						

- FP = count total x [0.65 + 0.01 x $\sum (F_i)$]
- F_i (i=1 to 14) are value adjustment factors (VAF) based on responses to the following questions:

- A system has 12 external inputs, 24 external outputs, fields 30 different external queries, manages 4 internal logical fi les, and interfaces with 6 different legacy systems (6 EIFs).All of these data are of average complexity and the overall system is relatively simple. Compute FP for the system.

- # Metrics for DESIGN
- Architectural Design Metrics
- Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. These metrics are "black box" in the sense that they do not require any knowledge of the inner workings of a particular software component.
- Card and Glass [Car90] define three software design complexity measures:
- **structural complexity, data complexity, and system complexity.**
- For hierarchical architectures (e.g., call-and-return architectures), *structural*
- *complexity* of a module *i* is defined in the following manner:

For hierarchical architectures (e.g., call-and-return architectures), *structural complexity* of a module i is defined in the following manner:

$$S(i) = f^2_{out}(i) \tag{30.2}$$

where $f_{out}(i)$ is the fan-out[8] of module i.

Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = \frac{v(i)}{[f_{out}(i) + 1]} \tag{30.3}$$

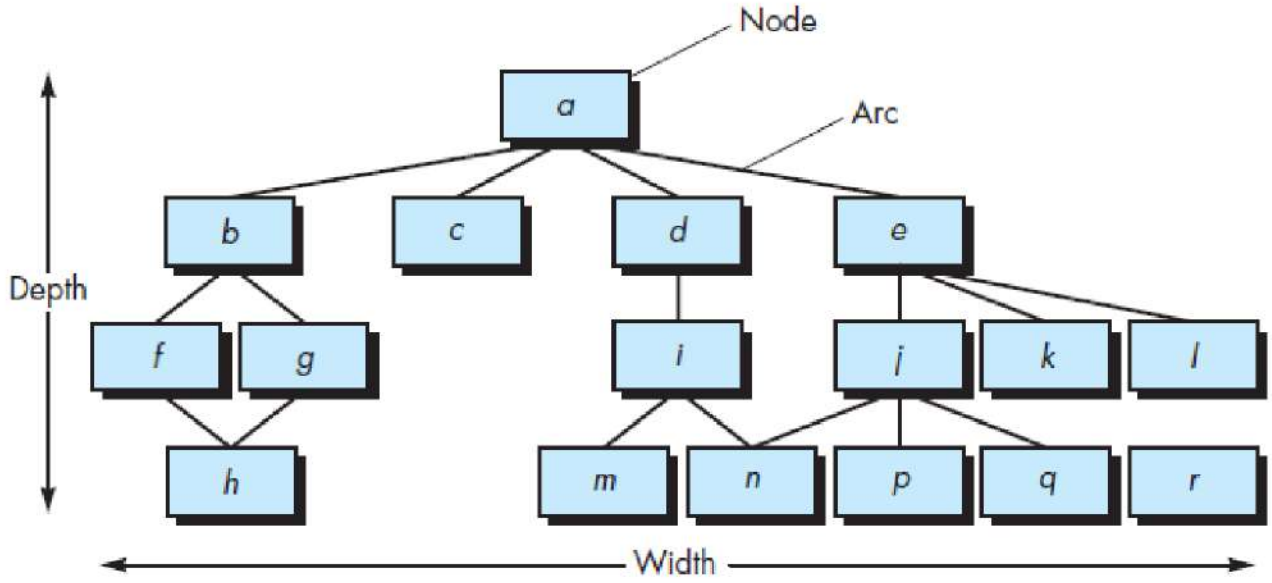
where $v(i)$ is the number of input and output variables that are passed to and from module i.

Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i) \tag{30.4}$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

- Fenton [Fen91] suggests a number of simple morphology (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to the call-and-return architecture the following metrics can be defined:
- Size = $n + a$
- where n is the number of nodes and a is the number of arcs.
- Size = 17 + 18 = 35
- Depth = longest path from the root (top) node to a leaf node. For the
- architecture shown in Figure , depth = 4.
- Width = maximum number of nodes at any one level of the architecture.
- For the architecture shown in Figure, width = 6.



The arc-to-node ratio, $r = a/n$, measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture.

For the architecture shown in Figure , $r = 18/17 = 1.06$.

- The U.S. Air Force Systems Command [USA87] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program.
- Using concepts similar to those proposed in IEEE Std. 982.1-2005 [IEE05], the Air Force uses information obtained from data and architectural design to derive a *design structure quality index* (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI [Cha89]:

- S 1= total number of modules defined in the program architecture
- S 2= number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S 2)
- S 3= number of modules whose correct function depends on prior processing
- S 4= number of database items (includes data objects and all attributes that define objects)
- S 5= total number of unique database items
- S 6= number of database segments (different records or individual objects)
- S 7= number of modules with a single entry and exit (exception processing is not considered to be a multiple exit)

- Once values *S* 1 through *S* 7 are determined for a computer program, the following intermediate values can be computed:
- *Program structure: D* 1, where *D* 1 is defined as follows:
- If the architectural design was developed using a distinct method (e.g., data flow–oriented design or object-oriented design), then *D* 1= 1, otherwise *D* 1= 0.

$$\textit{Module independence: } D_2 = 1 - \left(\frac{S_2}{S_1}\right)$$

$$\textit{Modules not dependent on prior processing: } D_3 = 1 - \left(\frac{S_3}{S_1}\right)$$

$$\textit{Database size: } D_4 = 1 - \left(\frac{S_5}{S_4}\right)$$

$$\textit{Database compartmentalization: } D_5 = 1 - \left(\frac{S_6}{S_4}\right)$$

$$\textit{Module entrance/exit characteristic: } D_6 = 1 - \left(\frac{S_7}{S_1}\right)$$

With these intermediate values determined, the DSQI is computed in the following manner:

$$\text{DSQI} = \Sigma\, w_i D_i$$

- where $i = 1$ to 6, $w\ i$ is the relative weighting of the importance of each of the intermediate values, and $\Sigma\ wi = 1$ (if all Di are weighted equally, then $w\ i = 0.167$).
- The value of DSQI for past designs can be determined and compared to a design that is currently under development.
- If the DSQI is significantly lower than average, further design work and review are indicated.
- Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

- METRICS FOR SOURCE CODE
- Halstead Metrics:
- According to Halstead's "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operand."
- Token Count
- In these metrics, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these basic symbols. These symbols are called as a token. The basic measures are
- n1 = count of unique operators.
- n2 = count of unique operands.
- N1 = count of total occurrences of operators
- N2 = count of total occurrence of operands.
- In terms of the total tokens used, the size of the program can be expressed as

$$N = N1 + N2.$$

- Halstead metrics are:
- **Program Volume (V)**
- The unit of measurement of volume is the standard unit for size "bits." It is the actual size of a program if a uniform binary encoding for the vocabulary is used.
- $V=N*\log_2 n$
- **Program Level (L)**
- The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size).
- $L=V*/V$
- **Potential Minimum Volume**
- The potential minimum volume V* is defined as the volume of the most short program in which a problem can be coded.
- $V* = (2 + n2*) * \log_2 (2 + n2*)$
- Here, n2* is the count of unique input and output parameters

Program Difficulty

The difficulty level or error-proneness (D) of the program is proportional to the number of the unique operator in the program.

$$D = (n1/2) * (N2/n2)$$

Programming Effort (E)

The unit of measurement of E is elementary mental discriminations.

$$E = V/L = D*V$$

Estimated Program Length

- According to Halstead, The first Hypothesis of software science is that the length of a well-structured program is a function only of the number of unique operators and operands.
- $N = N1 + N2$

And estimated program length is denoted by N^

- $N^{\wedge} = n1\log_2 n1 + n2\log_2 n2$

METRICS FOR MAINTENANCE

IEEE Std. 982.1-2005 [IEE05] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:

M_T = number of modules in the current release
F_c = number of modules in the current release that have been changed
F_a = number of modules in the current release that have been added
F_d = number of modules from the preceding release that were deleted in the current release

The software maturity index is computed in the following manner:

$$\text{SMI} = \frac{[M_T - (F_a + F_c + F_d)]}{M_T}$$

As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.