GANPAT UNIVERSITY

U. V. PATEL COLLEGE OF ENGINEERING

2CEIT502 SOFTWARE ENGINEERING

UNIT 6

CODING AND TESTING

Prepared by: Prof. Ravi Raval (Asst. Prof in C.E Dept. , UVPCE)

Unit 6: Coding and Testing

Contents:

6.1 Introduction

6.2 Coding Standards and Guidelines

6.3 Code Review

6.4 What is testing?

- 6.4.1 Error, Faults
- 6.4.2 Failure
- 6.4.3 Test Cases
- 6.4.4 Test Suits
- 6.4.5 Verification Vs. Validation

6.5 Testing Activities

- 6.5.1 Design Test Cases
- 6.5.2 Testing in Large Vs. Small

6.6 Black-Box Testing

6.7 White-Box Testing

6.8 Integration Testing

6.9 System Testing

Unit 6: Coding and Testing

6.1 Introduction

- Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.
- Test each module in isolation, After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.
- Over the years, the general perception of testing as monkeys typing in random data and trying to crash the system has changed. Now testers are looked upon as masters of specialized concepts, techniques, and tools.

Unit 6: Coding and Testing

6.2 Coding standards and Guidelines

- The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.
- Normally, A good software development organisation require their programmers to follow some well-defined and standard style of coding which is called their coding standard
- **Advantages** of following such **coding standards:**
  - A coding standard gives a uniform appearance to the codes written by different engineers.
  - It facilitates code understanding and code reuse.
  - It promotes good programming practices.
- Besides the **coding standards,** several **coding guidelines** are also prescribed by software companies. But, what is the **difference** between a coding guideline and a coding standard?
- Coding standard is compulsory to follow. During code review if standard violation is found, reject the code and recode it. Where as coding guidelines are general suggestions regarding the coding style to be followed.

Unit 6: Coding and Testing

6.2 Coding standards and Guidelines

<u>Coding standards:</u>

1. Rules for limiting the use of global variable.
2. Standard headers for different modules.

   Example: /*

   Module Name: Checkout option in E-Commerce

   Module Creation Date: 25/11/2011

   Author's Name: Mr. ABC DEF, Senior Programmer

   Modification history: <version number>

   Synopsis: <Brief details about the operation of module>

   List of function supported: fun1(), Fun2() with parameters

   Global variables accessed/modified: list of all such variables */
3. Naming conventions for global variables, local variables, and constant identifiers.
4. Conventions regarding error return values and exception handling.

Unit 6: Coding and Testing

6.2 Coding standards and Guidelines

<u>Coding Guidelines</u>

1. Do not use a coding style that is too clever or too difficult to understand
2. Avoid obscure side effects.
   - value of a global variable is changed.
   - some file I/O is performed obscurely.
   - modification to the parameters passed by reference.
3. Do not use an identifier for multiple purposes.

   Why? What are the consequences?
   - use of one identifier for multiple purpose leads to confusion.
   - Future enhancement will be more difficult.
4. Code Should be well documented (1 comment / 3 source code line)
5. Length of any function should not exceed 10 source code line.
6. Do not use GOTO Statement.

Unit 6: Coding and Testing

6.3 Code review

- Code review will be carried out when the module is successfully compiled (i.e. all syntax error is removed). Code reviews are extremely cost-effective strategies for eliminating coding errors and for producing high quality code.
- Normally, there are two following types of reviews are carried out:

(1) Code walkthrough (2) Code Inspection

(1) <u>Code Walkthrough</u>

The Main objective of code walkthrough is to discover the algorithmic and logical errors in the code.

- Give the compiled code to few development member before walk through meeting. Each member will select some test case, apply it on the code and note the traces of execution.
- In the meeting of code walkthrough they will discuss about their findings.
- There are some guidelines for code walkthrough however code walk through is based on personal experience, common sense and other subjective factors. Some guidelines are:

a) The team member of code walkthrough should consist of 3 to 7 members only.
b) Discuss about discovery of errors. Avoid discussion of how to solve it
c) Manager should not attend the meeting so engineers won't feel that they are being watched or evaluated.

Unit 6: Coding and Testing

6.3 Code review

(2) <u>Code Inspection</u>

- During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code walkthroughs.
- The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.
- Some Examples of classical programming errors:

1) Use of uninitialized variable.
2) Jump into loops.
3) Non-terminating loops (infinite loops)
4) Incompatible assignments.
5) Array indices out of bounds.
6) Improper storage allocation and de-allocation.
7) Mismatch between actual and formal parameter in procedure calls.
8) Use of incorrect logical operators or incorrect precedence among operators.
9) Improper modification of loop variables.
10) Comparison of equality of floating point values.
11) Dangling reference caused when the referenced memory has not been allocated.

(3) <u>Classroom Testing:</u> Clean room testing was pioneered at IBM. This type of testing relies heavily on walkthroughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. It is interesting to note that the term cleanroom was first coined at IBM by drawing analogy to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere

Unit 6: Coding and Testing

6.4 What is testing?

- Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction.

Terminologies (standardized by IEEE90):

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. Ex: Variable uninitialized, Div by 0.
- An **error** is the result of a mistake committed by a developer in any of the development activities. An error is also referred as **fault, bug** or a **defect**. Ex: Call made to a wrong function.
- A **failure** is a manifestation of an error. A failure of a program essentially denotes an incorrect behavior exhibited by the program during its execution. The reasons of failure can be in large number of ways. Some Random examples:
  - The result of computed program is 0, where as the correct result should be 10
  - A program crashed when giving some input. (e.g: Enter number: Hello)
  - A robot fails to avoid an obstacle and collides with it

Unit 6: Coding and Testing

6.4 What is testing?

- A **test case** is a triplet [I , S, R], where

I is the data input to the program under test,

S is the state of the program at which the data is to be input, and

R is the result expected to be produced by the program.

Example of test case for text editing software: is— [input: "abc", state: edit, result: abc is displayed] where different Mode can be edit, view, create and display.

- A **test suite** is the set of all test cases with which a given software product is tested.

6.4.5 Verification Vs. Validation

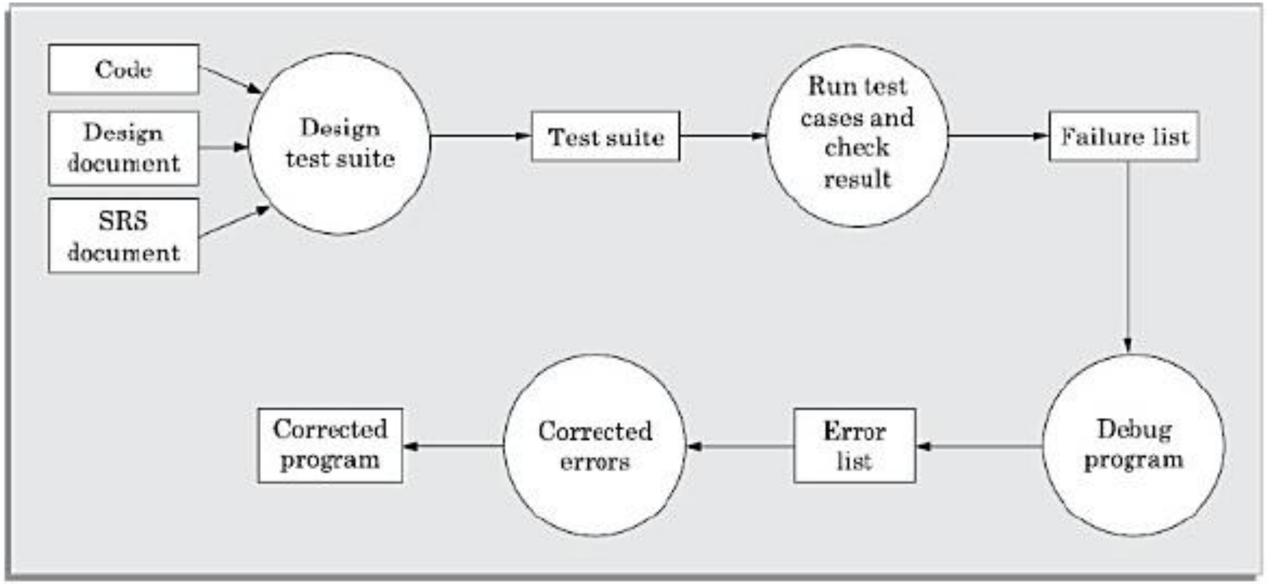
Verification	Validation
Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase.	validation is the process of determining whether a fully developed software conforms to its requirements specification
Primary techniques for verification includes review, simulation, formal verification and testing	validation techniques are primarily based on product testing
Unit testing, integration testing	System validation
It doesn't require whole software execution	It requires entire software execution
This activity carried our during the software development	This activity will be carried out after the software development.
In verification, we are trying to achieve phase containment of error. So that it incurs low cost and overhead.	Whereas here; not only we have to rework the design, but also to redo the relevant coding as well as the system testing activities, incurring higher cost.

Unit 6: Coding and Testing

6.5 Testing Activities

Testing Activities

1. Test Suite Design:
2. Running test cases and checking the results to detect failures
3. Debugging
4. Error correction



Unit 6: Coding and Testing

6.5 Testing Activities

6.5.1 Designing Test Cases

- When test cases are designed based on random input data, may of the test case do not contribute to the significance of the test suite. That means, they do not help in detecting any additional defects not already being detected by other test cases in the suite.
- Testing a system using a large collection of test cases that are selected at random doesn't guarantee that all of the errors in the system will be exposed.

Surprised … ? Let's see why don't it uncovers…?

Ex: A Code to find out MAX from two numbers

```
If ( x > y)     MAX=x;
else   MAX=x; //little typing mistake
```

Here, the test suite {(x=3,y=2);(x=2,y=3)} can detect the error.

Whereas, large test suite {(x=3,y=2); (x=4,y=3); (x=5,y=1); (x=10,y=9)} will not detect error. Moreover larger test suite incurs high cost, efforts and time.

- A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contrast to testing using some random input values.
- Two main approaches to systematically design test cases

1) Black-Box Approach (a.k.a functional testing)
2) White-Box Approach (a.k.a structural testing)

Unit 6: Coding and Testing

6.5 Testing Activities

6.5.2 Testing in small vs. Testing in Large

A software product is normally tested in three level or stages:

1) Unit Testing } Testing in small
2) Integration Testing } Testing in Large
3) System Testing

Unit Testing:

- Unit testing will be carried out in coding phase itself as soon as your coding done. Unit test will be performed by developer.
- To perform unit test you have to provide necessary environment (relevant code)

- The procedures belonging to other modules that the module under test calls.
  - Non-local data structures that the module accesses.
  - A procedure to call the functions of the module under test with appropriate parameters.

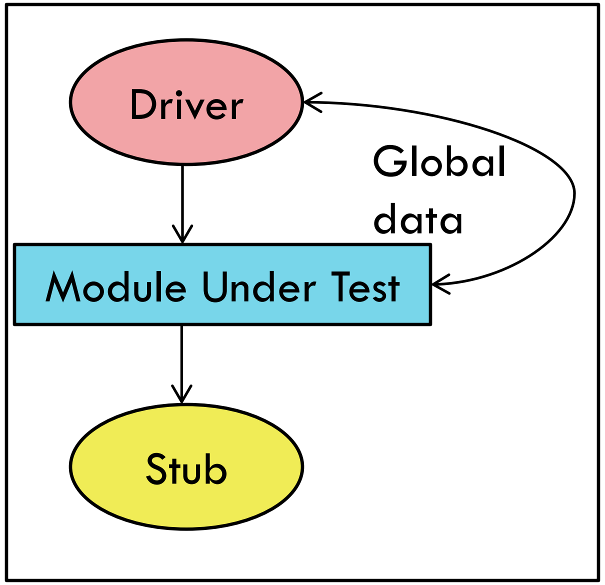
Unit 6: Coding and Testing

6.5 Testing Activities

6.5.2 Testing in small vs. Testing in Large

Stub: A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified.

Driver: A driver module should contain the non-local data Structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.



Unit 6: Coding and Testing

6.6 Black Box Testing

- In 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.
- Approaches to design black box test case:
  1. Equivalent Class Partitioning
  2. Boundary value Analysis

(1) Equivalent Class Partitioning:

The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.

- **Guidelines:**

1) Prepare 1 valid class of input (For Ex. [1,10]) and 2 invalid class of input (For Ex. [-∞,0] , [11, +∞]).
2) If the input data assumes values from a set of discrete members of some domain, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is U -{A,B,C}, where U is the universe of possible input values.

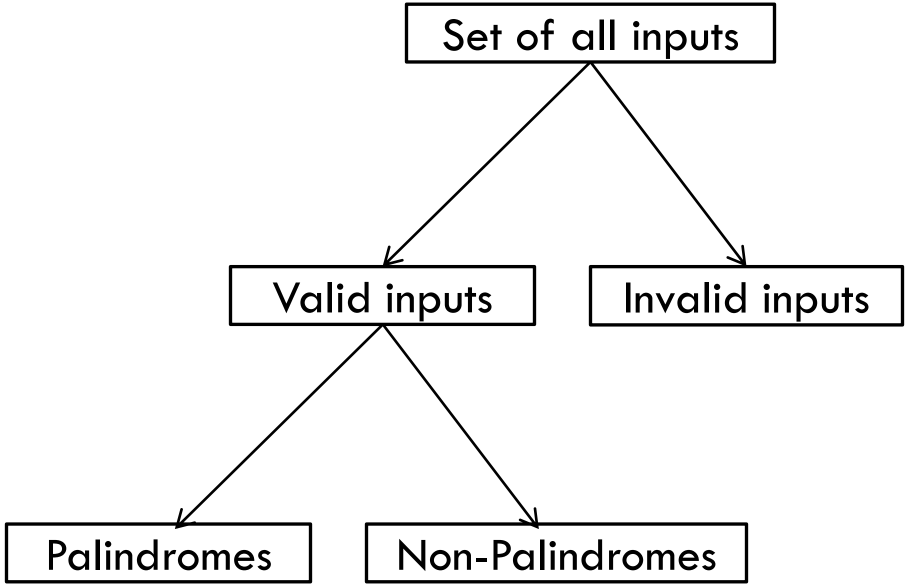
Unit 6: Coding and Testing

6.6 Black Box Testing

Example: Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

Solution:

{ abc, aba, abcdef }



Unit 6: Coding and Testing

6.6 Black Box Testing

(2) Boundary Value Analysis:

- Boundary value analysis means limiting the range of values to be provided.
- Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.
- For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Example: For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

Answer:

There are three equivalence classes

- The set of negative integers,
- the set of integers in the range of 0 and 5000,
- and the set of integers larger than 5000.

- The boundary value-based test suite is: {0,-1,5000,5001}.

Unit 6: Coding and Testing

6.7 White Box Testing

Basic Concepts:

- A White box testing strategy is either be fault based or coverage based.

Fault Based Testing	**Coverage Based Testing**
A fault-based testing strategy targets to detect certain types of faults.	A coverage-based testing strategy attempts to execute (or cover) certain elements of a program.
Ex: mutation testing	Ex: Statement coverage, branch coverage, multiple condition coverage and path coverage

Testing Criterion for coverage-based testing

The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

Stronger Testing vs. Weaker Testing

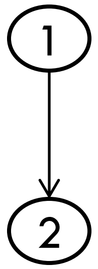
A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

Unit 6: Coding and Testing

6.7 White Box Testing

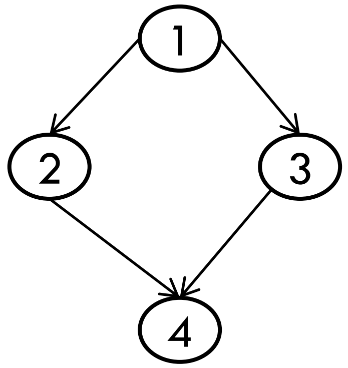
Stronger

```
a=5;
b=a*2-1
```



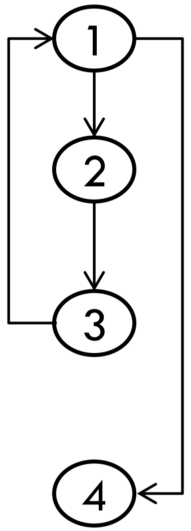
Weaker

```
if(a>b)
  c=3;
else c=5;
c=c*c
```



Complementary

```
while(a>b){
            b=b-1;
            b=b*a;
   }  c=a+b;
```



Unit 6: Coding and Testing

6.7 White Box Testing

Statement Coverage

- The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.
- The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.
- It is obvious that without executing a statement, it is difficult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement- coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique.

Unit 6: Coding and Testing

6.7 White Box Testing

Statement Coverage

Example: For a given program of Euclid's GCD computation, what test case will be required to find out that every statement will be covered (execute at least once)

Code:

```
int computeGCD(x,y)
int x,y;
{
          while (x != y)
          {
                    if (x>y) then
                              x=x-y;
                    else
                              y=y-x;
          }
return x;
}
```

By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y =4)}, all statements of the program would be executed at least once.

Unit 6: Coding and Testing

6.7 White Box Testing

Branch Coverage

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn.
- Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

```
int computeGCD(int x,int y)
{
      while (x != y)
      {
            if (x>y) then
                  x=x-y;
            else
                  y=y-x;
      }
return x;
}
```

The test suite {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)} achieves branch coverage.

Unit 6: Coding and Testing

6.7 White Box Testing

Multiple Condition Coverage:

- In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3).
- It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage.
- Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Example: Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage

if(temperature>150 || temperature>50) // **Second condition should be < 50**

setWarningLightOn(); // **Because, then and only test suite (temperature=160,60) will achieve branch coverage.**

Unit 6: Coding and Testing

6.7 White Box Testing

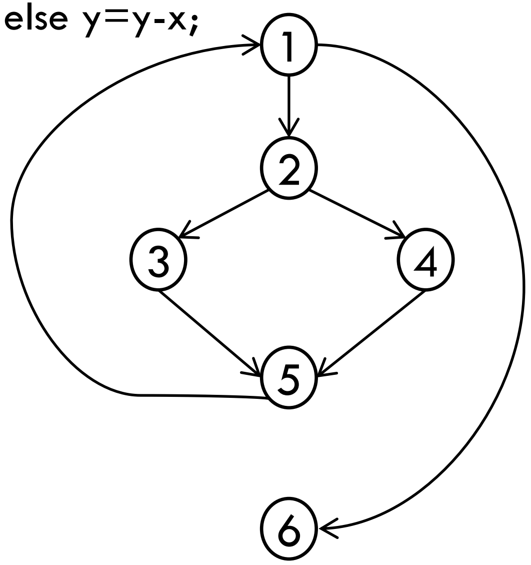
Path Coverage:

- A test suite achieves path coverage if it executes each linearly independent paths (or basis paths) at least once.
- A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control Flow Graph: A control flow graph describes the sequence in which the different instructions of a program get executed.

```
int computeGCD(int x,int y){
1           while (x != y){
2                       if (x>y) then
3                               x=x-y;
4                       else y=y-x;
5           }
6 return x;
}
```

Solution



Unit 6: Coding and Testing

6.7 White Box Testing

Path: A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program.

- A program normally has more than one terminal node because of multiple return or exit type of statements. So one cannot write test cases to cover all the paths as there can be infinite no. of paths can be exist in a program with presence of loop.
- If you try to cover all paths then your test case will be infinitely large. So path coverage doesn't focus to cover all paths but only a subset of paths called linearly independent path (or base path).

Linearly independent set of paths (Base path set)

A set of paths for a given program is called ***linearly independent set of paths***, if each path in the set introduces at least one new edge that is not included in any other path in the set.

Unit 6: Coding and Testing

6.7 White Box Testing

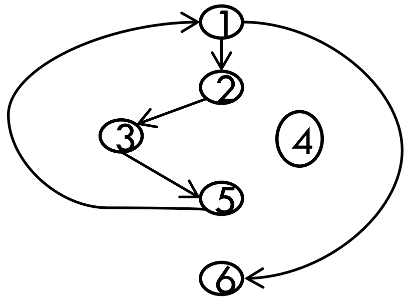
<u>McCabe's Cyclomatic Complexity Metric</u>

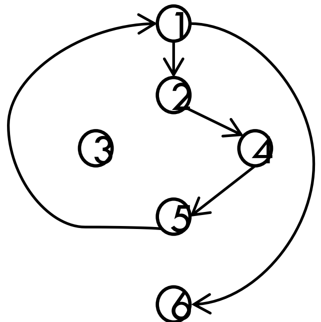
- McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program.
- There are three different ways to compute the cyclomatic complexity.

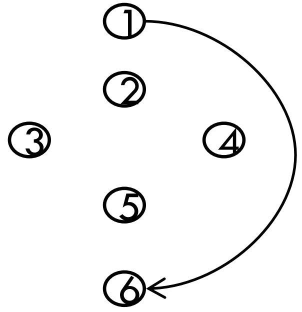
<u>Method 1:</u> If G is control flow graph (CFG) of a program then cyclomatic complexity

V(G) = E – N + 2 (where, N is no. of nodes and E is no. of edges in CFG)

For ex. Cyclomatic complexity for previous figure V(G) = 7 – 6 + 2 = 3







Unit 6: Coding and Testing

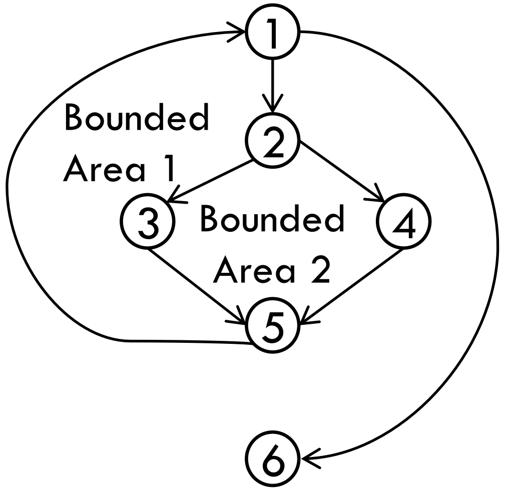
6.7 White Box Testing

McCabe's Cyclomatic Complexity Metric

Method 2: is based upon visual inspection of CFG

- Here, cyclomatic complexity V(G)=total no. of non-overlapping bounded areas+1.
- In any given CFG, any region enclosed by nodes and edges can be called as a bounded area.
- Bounded are means closed area.

Here, in previous example; the total no. of bounded areas were 2.
So,
V(G)=total no. of non-overlapping bounded areas
V(G)=2+1=3



Unit 6: Coding and Testing

6.7 White Box Testing

McCabe’s Cyclomatic Complexity Metric

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program.
If N is the number of decision and loop statements of a program, then the McCabe’s metric is equal to N + 1.

So here in this given program:

```
int computeGCD(int x,int y)
{         while (x != y)
          {         if (x>y) then
                              x=x-y;
                    else y=y-x;
          }
          return x;
}
```

Solution:

No. of decision making statements are: 1

- If (x > y)

No. of Looping statements are: 1

- While (x != y)

Hence, N=1+1=2

So V(G) =N+1 =2+1 =3

Unit 6: Coding and Testing

6.7 White Box Testing

- **Steps to carry out path coverage-based testing:**

1. Draw control flow graph for the program.
2. Determine the McCabe's metric V(G).
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. Repeat.

Test using a randomly designed set of test cases. Perform dynamic analysis to check the path coverage achieved until at least 90 per cent path coverage is achieved.

- **Uses of McCabe's Cyclomatic complexity metric:**

1. Estimation of structural complexity of code.
2. Estimation of testing effort.
3. Estimation of program reliability

Unit 6: Coding and Testing

6.7 White Box Testing

Mutation testing:

- All testing techniques discussed so far are coverage-based testing techniques.
- Where as, mutation testing is fault-based testing technique.
- In mutation based technique, first test the program using any white box testing strategies. The idea of mutation based technique is to make a few arbitrary changes (called mutated) to a program at a time. The change is effect is called mutant.
- For ex: one mutation operator deletes a statement and then after executes the program and if the mutated program results same as the original program then both programs are called equivalent program.
- If a mutant is failed at least one test case to pass then its called dead. Since the error introduced by the mutation operator has successfully been detected by the test suite. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant.
- This process can be automated that is its great advantage.
- A major pitfall of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Unit 6: Coding and Testing

6.8 Integration Testing

- Integration testing will be carried out-once all module has been unit tested.
- A major pitfall of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated. i.e., there are no errors in parameter passing, when one module invokes the functionality of another module.
- During this testing, combine modules step by step and perform test.

  The structure chart will help you to know the order of different modules call each other. So by using this chart you can develop the integration test.
- Any one or mixture of following approaches can be used for integration test:

1) Big-bang approach to integration testing
2) Top-down approach to integration testing
3) Bottom-up approach to integration testing
4) Mixed (also called sandwiched) approach to integration testing

Unit 6: Coding and Testing

6.8 Integration Testing

1) **Big bang approach**
- In this approach, all the modules making up a system are integrated in a single step.
- Advantage: Can be used for only small system.
- Problem: once you find an error, its difficult to localize (locating) the error. Plus it would be very expensive to fix the errors. So its bad for big software.

2) **Bottom up approach:**
- Large software are often broken into subsystems.
- first the modules for the each subsystem are integrated and tested.
- REPEAT the process till all subsystem will be integrated and tested independently.
- Advantage: several disjoint subsystem can be tested simultaneously.
- Advantage: Low level modules are exercised thoroughly in each stage.
- Disadvantage: Complexity may occur when a software has large number of small subsystem at the same level. So in this case it will be same as big bang.

Unit 6: Coding and Testing

6.8 Integration Testing

3) **Top down approach:**

- Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested.

Advantage: it requires writing only stubs and stubs are simpler to write then drivers

A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

4) **Mixed approach:**

- The mixed or sandwiched approach combines top down and bottom up approaches. So It overcomes all shortcomings of top down and bottom up.
- Here, testing can start as and when modules becomes available after unit test.
- Disadvantage is only that you need to design both stub and drivers

Unit 6: Coding and Testing

6.9 System Testing

- Will be carried out when all units has integrated together and tested.
- System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.
- There are three types of system test depending upon who carries out:

1. **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organisation.
2. **Beta Testing:** Beta testing is the system testing performed by a select group of friendly customers.
3. **Acceptance Testing:** Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

In above all methods, test cases may be the same, but the difference is who design them.

- Before a fully integrated system is accepted for system testing, smoke testing is performed. Smoke testing is done to check whether at least the main functionalities of the software are working properly

Unit 6: Coding and Testing

6.9 System Testing

6.8.1 Smoke Testing:

- Smoke test will be carried out before system testing.
- The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing.
- How to identify basic functionalities are working or not?
  - for a library automation system, the smoke tests may check whether books can be created and deleted,
  - whether member records can be created and deleted,
  - and whether books can be loaned and returned.

6.8.2 Performance Testing:

Performance testing is carried out to check whether the system meets the nonfunctional requirements identified in the SRS document.

1) **Stress Testing (a.k.a endurance testing):**
   - Stress testing evaluates system performance when it is stressed for short periods of time
   - Stress the capability of software by supplying abnormal and illegal inputs.
   - What is to be testing in this? Input data volume, input data rate, processing time, utilization of memory etc.
2) **Compatibility Testing**
   - requires for those software using external interface (For ex. Database, servers)

Unit 6: Coding and Testing

6.9 System Testing

3) **Volume Testing**
   Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.
   For Ex. For compiler; symbol table overflows when large program is compiled.
4) **Configuration Testing**
   To test the system behavior in various hardware and software configuration
   The system is configured in each of the required configurations (single user, multi user) and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.
5) **Regression Testing**
   This test will be essential when a software is maintained to fix some bugs or enhance functionality, performance
6) **Recovery Testing**
   Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc.
   For ex. Disconnect printer cause system hangs, improper shutdown cause data loss
7) **Maintenance Testing**
   Testing the diagnostic programs (e.g system restore, disk defragment)
8) **Documentation Testing**
   Check whether user manual, maintenance manual and technical manual exist.
9) **Usability testing**
   Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface.
   During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested.

Unit 6: Coding and Testing

6.9 System Testing

6.8.3 <u>Security Testing</u>

- Security testing is essential for software that handle or process confidential data that is to be guarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers.
- Over the last few years, a large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports etc.

6.8.4 <u>Error Seeding</u>

- Sometimes customer specifies no. of allowed errors per line of source code.
- The error seeding technique can be used to estimate the number of residual errors in a software
- In this technique, some artificial errors are introduced (seeded) into the program
- The number of these seeded errors that are detected in the course of standard testing is determined These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program:

Unit 6: Coding and Testing

6.9 System Testing

- The number of errors remaining in the product.
- The effectiveness of the testing strategy

Let, N=number of defects in the system
n=these defects found by testing
S=total number of seeded defects
s=these defects found during testing. Therefore, we get $\frac{n}{N} = \frac{s}{S}$

$$or, \quad N = S \times \frac{n}{s}$$

Defects still remaining in the program after testing can be given by:

$$\mathrm{N} - n = n \times \frac{(S-1)}{s}$$

- This methods works well only if you seeded errors occurs actually occurs frequently. So it is difficult to predict type of errors that exist in a software. You can only estimate such errors by analyzing historical data collected from similar projects.

============== END OF CHAPTER =========== END OF SYLLABUS ==============