UNIT 4

CODING

Coding- 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. The programmers adhere to standard and well defined style of coding which they call their coding standard. The main advantages of adhering to a standard style of coding are as follows:

- A coding standard gives uniform appearances to the code written by different engineers
- It facilitates code of understanding.
- Promotes good programming practices.

For implementing our design into a code, we require a good high level language. A programming language should have the following features:

Characteristics of a Programming Language

- **Readability:** A good high-level language will allow programs to be written in some ways that resemble a quite-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.

- **Portability:** High-level languages, being essentially machine independent, should be able to develop portable software.

- **Generality:** Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages.

- **Brevity:** Language should have the ability to implement the algorithm with less amount of code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.

- **Error checking:** Being human, a programmer is likely to make many mistakes in the development of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.

- **Cost:** The ultimate cost of a programming language is a function of many of its characteristics.

- **Familiar notation:** A language should have familiar notation, so it can be understood by most of the programmers.

- **Quick translation:** It should admit quick translation.

- **Efficiency:** It should permit the generation of efficient object code.

- **Modularity:** It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.

- **Widely available:** Language should be widely available and it should be possible to provide translators for all the major machines and for all the major operating systems.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

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 information contained in the headers of different modules should be standard for an organization. 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.

3. **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.

4. **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.

2. **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.

3. **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. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

   - Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.

- Use of variables for multiple purposes usually makes future enhancements more difficult.

4. **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.

5. **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.

6. **Do not use goto statements:** Use of goto statements makes a program unstructured and very difficult to understand.

What is Coding Standards and Guidelines?

Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review.

Purpose of Having Coding Standards

The following are the purpose of having Coding Standards:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It improves readability, and maintainability of the code and it reduces complexity also.
- It helps in code reuse and helps to detect errors easily.
- It promotes sound programming practices and increases the efficiency of the programmers.

Coding Standards in Software Engineering

Some of the coding standards are given below:

1. **Limited use of globals:** These rules tell about which types of data that can be declared global and the data that can't be.
2. **Standard headers for different modules:** For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:
    - Name of the module
    - Date of module creation
    - Author of the module
    - Modification history
    - Synopsis of the module about what the module does
    - Different functions supported in the module along with their input output parameters
    - Global variables accessed or modified by the module

3. **Naming conventions for local variables, global variables, constants and functions:**
   - Meaningful and understandable variables name helps anyone to understand the reason of using it.
   - Local variables should be named using camel case lettering starting with small letter (e.g. localData) whereas Global variables names should start with a capital letter (e.g. GlobalData). Constant names should be formed using capital letters only (e.g. CONSDATA).
   - It is better to avoid the use of digits in variable names.
   - The names of the function should be written in camel case starting with small letters.
   - The name of the function must describe the reason of using the function clearly and briefly.
4. **Indentation:** Proper indentation is very important to increase the readability of the code. For making the code readable, programmers should use White spaces properly. Some of the spacing conventions are given below:
   - There must be a space after giving a comma between two function arguments.
   - Each nested block should be properly indented and spaced.
   - Proper Indentation should be there at the beginning and at the end of each block in the program.
   - All braces should start from a new line and the code following the end of braces also start from a new line.
5. **Error return values and exception handling conventions**: All functions that encountering an error condition should either return a 0 or 1 for simplifying the debugging.

Coding Guidelines in Software Engineering

Coding guidelines give some general suggestions regarding the coding style that to be followed for the betterment of understandability and readability of the code.

Some of the coding guidelines are given below :

1. Avoid using a coding style that is too difficult to understand: Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.
2. Avoid using an identifier for multiple purposes: Each variable should be given a descriptive and meaningful name indicating the reason behind using it. This is not possible if an identifier is used for multiple purposes and thus it can lead to confusion to the reader. Moreover, it leads to more difficulty during future enhancements.
3. Code should be well documented: The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.
4. Length of functions should not be very large: Lengthy functions are very difficult to understand. That’s why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.
5. Try not to use GOTO statement: GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.

Advantages of Coding Guidelines

1. Coding guidelines increase the efficiency of the software and reduces the development time.

2. Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.

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. These two types code review techniques are code inspection and code walk through.

Code Walk Throughs

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. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present. Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following:

- 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.

Code Inspection

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

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

- Use of uninitialized variables.
- Jumps into loops.
- Nonterminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- 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 vari

TESTING

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.

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.

Verification Vs Validation

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

Design of Test Cases

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible. Actually, if test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered. Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

```
if (x>y)
        max = x;
else
        max = x;
```

For the above code segment, the test suite, **{(x=3,y=2);(x=2,y=3)}** can detect the error, whereas a larger test suite **{(x=3,y=2);(x=4,y=3);(x=5,y=1)}** does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

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.

In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module:

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

Modules are required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in fig. 19.1. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

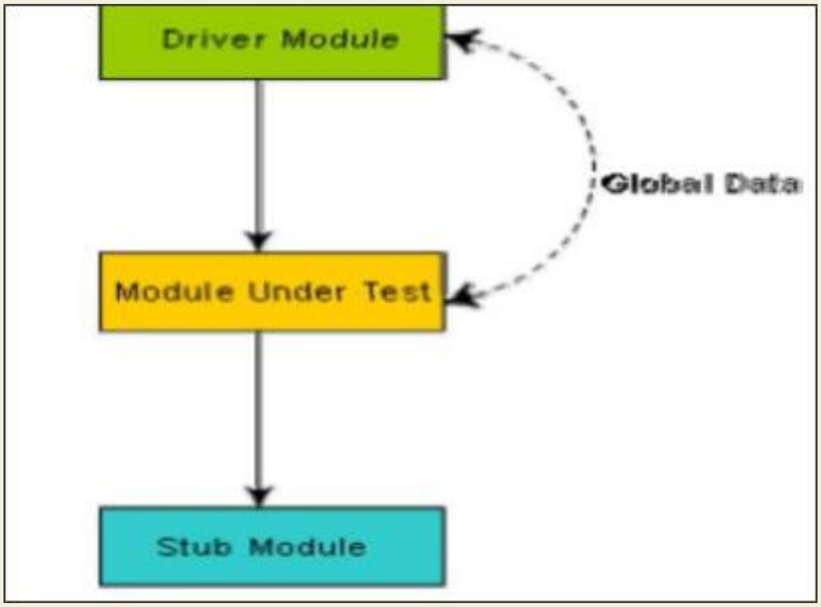


Fig. 19.1: Unit testing with the help of driver and stub modules

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 portioning
- 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 sys1.tem 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.

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}.

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 figA

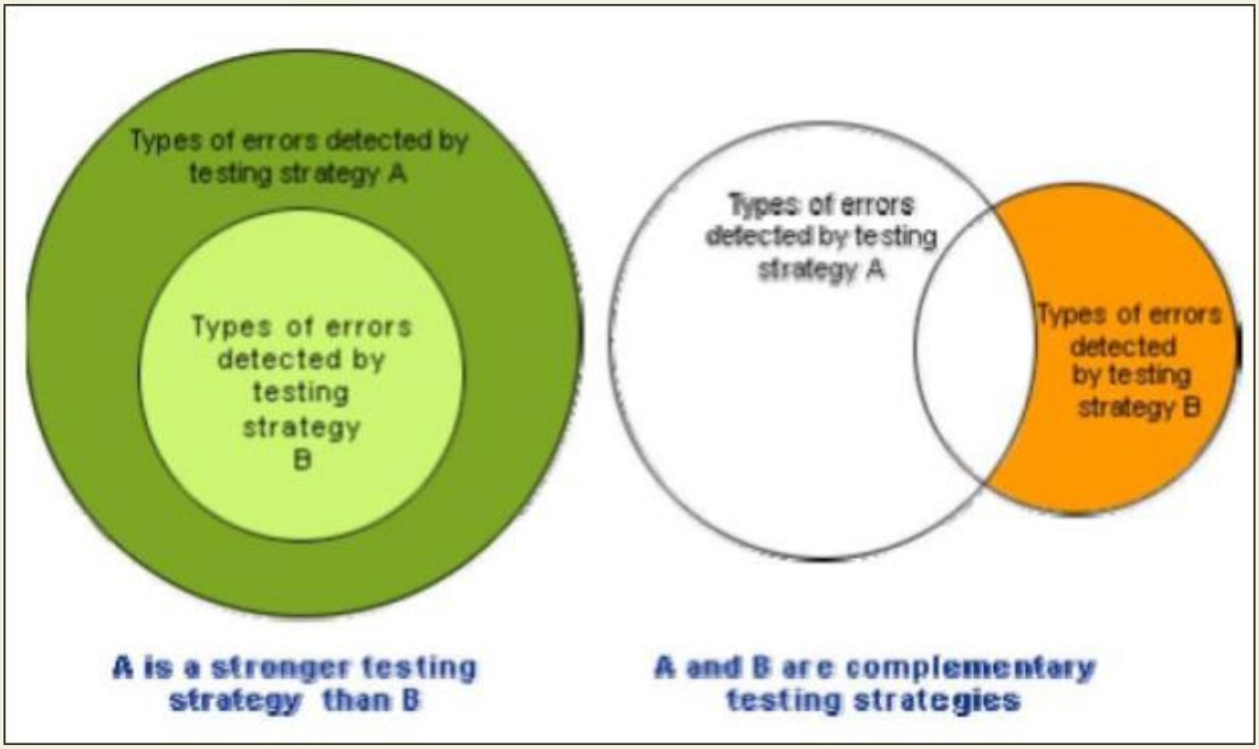

Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
A is a stronger testing strategy than B
A and B are complementary testing strategies

Fig A Stronger and 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. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid’s GCD computation algorithm:

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

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed at least once.

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.

It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid’s GCD computation algorithm, the test cases for branch coverage can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

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. Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

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. The different numbered statements serve as nodes of the control flow graph (as shown in fig. 20.2). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig. 20.2 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid’s GCD computation algorithm can be drawn as shown in fig. 20.3.

Sequence:

```
a=5;

b = a*2-1;
```

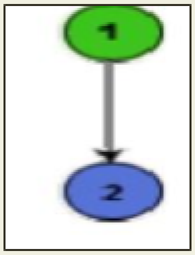


Fig. 20.2 (a): CFG for sequence constructs

Selection:

```
if (a>b)

        c = 3;
```

```
else
            c =5;
 c=c*c;
```

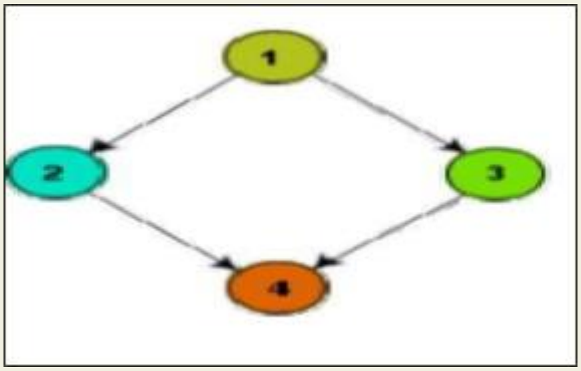


Fig. 20.2 (b): CFG for selection constructs

Iteration :

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

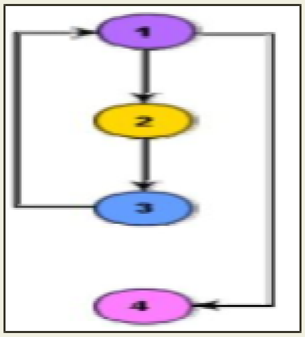
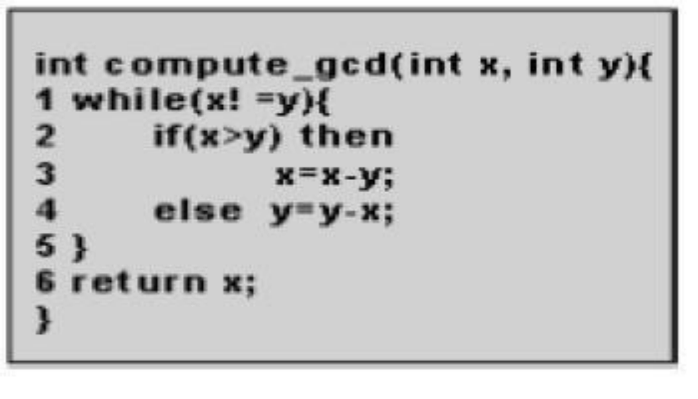


Fig. 20.2 (c): CFG for and iteration type of constructs

EUCLID’S GCD Computation Algorithm



```
int compute_gcd(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;
}
```

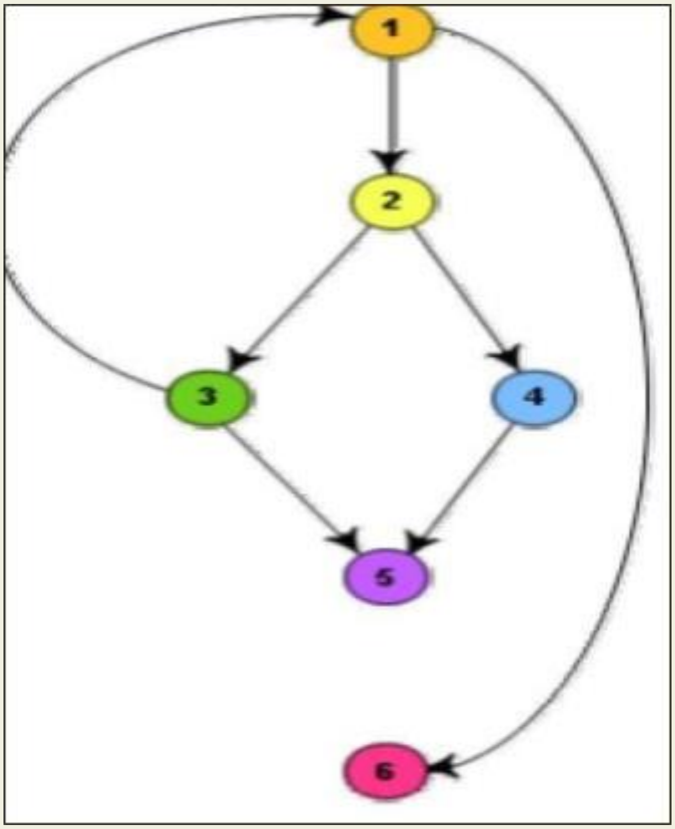


Fig. 20.3: Control flow diagram

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 Approaches

The following are some of the approaches popularly adopted by programmers for debugging.

Brute Force Method:

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

Backtracking:

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

Cause Elimination Method:

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program Slicing:

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

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 mistake 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.

Program Analysis Tools

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc. We can classify these into two broad categories of program analysis tools:

- Static Analysis tools
- Dynamic Analysis tools
- Static program analysis tools

Static Analysis Tool is also a program analysis tool. It assesses and computes various characteristics of a software product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

- Whether the coding standards have been adhered to?
- Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walk throughs and code inspections might be considered as static analysis methods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

***Dynamic program analysis tools* -** Dynamic program analysis techniques require the program to be executed and its actual behavior recorded. A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces). The instrumented code when executed allows us to record the behavior of the software for different test cases. After the software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool caries out a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete test suite for the program. For example, the post execution dynamic analysis report might provide data on extent statement, branch and path coverage achieved.

Normally the dynamic analysis results are reported in the form of a histogram or a pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily and provides evidence that thorough testing has been done. The dynamic analysis results the extent of testing performed in white-box mode. If the testing coverage is not satisfactory more test cases can be designed and added to the test suite. Further, dynamic analysis results can help to eliminate redundant test cases from the test suite.

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
- Bottom- up approach
- Top-down 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 Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

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.

Phased Vs. Incremental Testing

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partial system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system in the incremental testing approach since it is known that the error is caused by addition of a single module. In fact, big bang testing is a degenerate case of the phased integration testing approach.

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 test tests the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance test tests 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

Stress Testing -Stress testing is also known as ***endurance testing***. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multi programmed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours. For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

Volume Testing-It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations. For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

Configuration Testing - This is used to analyze system behavior in various hardware and software configurations specified in the requirements. Sometimes systems are built in variable configurations for different users. For instance, we might define a minimal system to serve a single user, and other extension configurations to serve additional users. The system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.

Compatibility Testing -This type of testing is required when the system interfaces with other types of systems. Compatibility aims to check whether the interface functions perform as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression Testing -This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

Recovery Testing -Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as applicable and discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance Testing- This testing addresses the diagnostic programs, and other procedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation Testing- It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance.

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, report formats, and other aspects relating to the user interface requirements are tested.

Regression Testing

Regression testing does not belong to either unit test, integration test, or system testing. Instead, it is a separate dimension to these three forms of testing. The functionality of regression testing has been discussed earlier.

Unit Testing

Unit Testing is a software testing technique by means of which individual units of software i.e. group of computer program modules, usage procedures and operating procedures are tested to determine whether they are suitable for use or not. It is a testing method using which every independent modules are tested to determine if there are any issue by the developer himself. It is correlated with functional correctness of the independent modules.

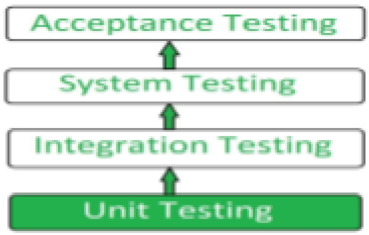
Unit Testing is defined as a type of software testing where individual components of a software are tested.

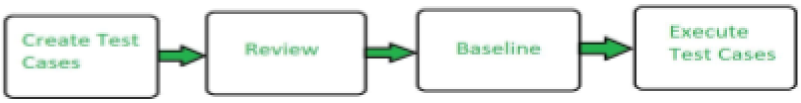
Unit Testing of software product is carried out during the development of an application. An individual component may be either an individual function or a procedure. Unit Testing is typically performed by the developer.

In SDLC or V Model, Unit testing is first level of testing done before integration testing. Unit testing is such type of testing technique that is usually performed by the developers. Although due to reluctance of developers to tests, quality assurance engineers also do unit testing.

Objective of Unit Testing:

1. To isolate a section of code.
2. To verify the correctness of code.
3. To test every function and procedure.
4. To fix bug early in development cycle and to save costs.
5. To help the developers to understand the code base and enable them to make changes quickly.
6. To help for code reuse.





Types of Unit Testing:

There are 2 types of Unit Testing: Manual, and Automated.

Unit Testing Tools:

1. Jtest
2. Junit
3. NUnit
4. EMMA
5. PHPUnit

Advantages of Unit Testing:

•Unit Testing allows developers to learn what functionality is provided by a unit and how to use it to gain a basic understanding of the unit API.

•Unit testing allows the programmer to refine code and make sure the module works properly.

Validation and System Testing

At the end of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected and now validation testing is performed. Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements.

After each validation test case has been conducted, one of two possible condition exist:

1. The function or performance characteristics conform to specification and are accepted or
2. a deviation from specification is uncovered and a deficiency list is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery.

Alpha and Beta Testing:

It is virtually impossible for a software developer to foresee how the customer will really use a program:

· Instructions for use may misinterpreted.

· strange combinations of data may be regularly used

· output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. If software is developed as a product to be used by many customers, it is impractical to perform acceptance tests with each one. alpha and beta tests are used to uncover errors that only the end-user seems able to find.

The Alpha Test is conducted at the developer’s site by a customer. The software is used in a natural setting with the developer “looking over the shoulder” of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The Beta test is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base .

System Testing:

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. System Testing is basically performed by a testing team that is independent of the development team that helps to test the quality of the system impartial. System Testing is carried out on the whole system in the context of either system requirement specifications or functional requirement specifications or in the context of both. System testing tests the design and behavior of the system and also the expectations of the customer.

Types of System Testing:

· Performance Testing: Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application

. · Load Testing: Load Testing is a type of software testing which is carried out to determine the behavior of a system or software product under extreme load.

·Stress Testing: Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads.

· Scalability Testing: Scalability Testing is a type of software testing which is carried out to check the performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

Desk Check

What is a Deskcheck?

- **Deskcheck** is an **informal review** process where a software developer **personally reviews** their own code or design to find errors **before** formal testing or peer review.
- It involves **self-review** and **manual inspection** of source code, logic, or documents.

Purpose of Deskcheck

- To **identify and fix errors early**, reducing bugs in later stages.
- To **improve the quality** of the code/design.
- To **save time and cost** by catching mistakes early.
- To ensure that code or design is **complete**, **consistent**, and **correct**.

Activities in Deskcheck

- Reviewing **logic errors**, **syntax errors**, and **design flaws**.
- Verifying that all **requirements** are properly implemented.
- Ensuring **proper coding standards** and **best practices** are followed.
- Checking for **optimization opportunities**.

Characteristics of Deskcheck

- **Informal**: No formal documentation or meeting is required.
- **Individual**: Usually performed by the developer themselves.
- **Quick**: Saves time compared to formal reviews.
- **Flexible**: Can be done at any point during development.

Benefits of Deskcheck

- Catches simple mistakes early.
- Improves the developer’s own understanding of their work.
- Reduces workload during formal review stages.
- Leads to cleaner, more efficient code/design.

Limitations of Deskcheck

- **Subjective**: Depends on the developer’s skill level.
- **Not exhaustive**: May miss some deeper or complex errors.
- **No formal record**: Issues identified may not be documented.

Coding Frameworks in Software Engineering

1. Definition & Purpose

- A **coding framework** is a reusable, semi-complete software skeleton that provides **standard structures**, **libraries**, and **APIs** to accelerate development.
- **Purposes:**
  - Enforce consistent **architecture** and **coding conventions**
  - Provide common **utility functions** (e.g., logging, error handling, data access)
  - Abstract away low-level boilerplate so teams focus on business logic
  - Improve **maintainability**, **testability**, and **scalability**

2. Key Characteristics

- **Modularity:** Well-defined components or modules you can plug in/out.
- **Configurability:** Settings or conventions (e.g., XML/JSON/YAML) to tailor behavior.
- **Convention over Configuration:** Sensible defaults to reduce explicit setup.
- **Extensibility:** Hooks or plugin points for custom extensions.
- **Documentation & Community:** Clear guides and active support for best practices.

3. Common Framework Types

1. **Web Application Frameworks**
   - MVC-style: **Django** (Python), **Ruby on Rails**, **ASP.NET MVC**, **Spring MVC**
   - Single-Page App: **Angular**, **React (with Redux)**, **Vue.js**
2. **API / Microservice Frameworks**
   - **Express** (Node.js), **FastAPI** (Python), **Spring Boot** (Java), **ASP.NET Core**
3. **Data-Access / ORM Frameworks**
   - **Hibernate** (Java), **Entity Framework** (.NET), **SQLAlchemy** (Python), **ActiveRecord** (Rails)
4. **Mobile App Frameworks**
   - **Flutter**, **React Native**, **Ionic**, **Xamarin**
5. **Testing Frameworks**
   - **JUnit/TestNG** (Java), **pytest** (Python), **NUnit** (.NET), **Jest/Mocha** (JavaScript)

4. Benefits

- **Speed of Development:** Ready-to-use components shorten time to market.
- **Consistency:** Unified patterns across modules and teams.
- **Quality & Reliability:** Battle-tested libraries reduce defects.
- **Maintainability:** Clear project structure and separation of concerns.
- **Community & Ecosystem:** Access to plugins, extensions, and shared solutions.