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.

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.

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.

Error checking: Being human, a programmer is likely to make many mistakes in the development of a computer program.

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.

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

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.

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.

Software Documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual etc. are also developed as part of any software engineering process.

Different types of software documents can be broadly classified into the following:

I. Internal Documentation:

This is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code.

II. External Documentation:

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

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:

I. **Failure:** This is a manifestation of an error (or defect, bug).

II. **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.

III. **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.

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.

BLACK BOX TESTING

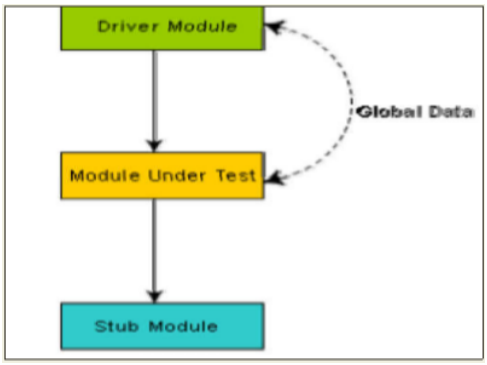
Testing in the Large VS Testing in the Small

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.

Modules are required to provide the necessary environment (which either call or are called by the module under test) which 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 the figure below. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly-simplified behavior. 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.



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:

I. Equivalence Class Partitioning

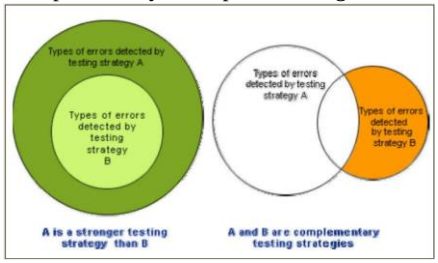
In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class

II. 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.

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 stronger and complementary concepts of testing are schematically illustrated in the figure below.



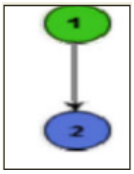
Control Flow Graph (CFG)

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 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. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown in the figure below:

Sequence:

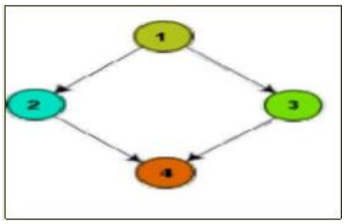
```
a=5;
b = a*2-1;
```



CFG for sequence constructs

Selection:

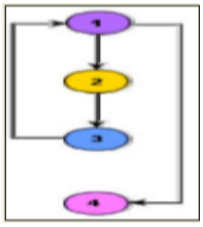
```
If (a>b)
        c= 3;
    else
            c=5;
    c=c*c;
```



CFG for selection constructs

Iteration:

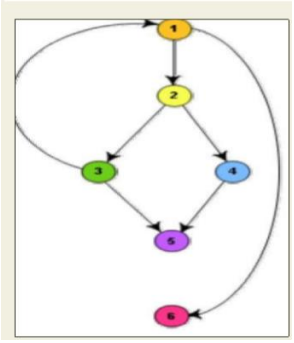
```
While (a>b)
{
        b=b-1;
        b=b*a;
}
C=a+b;
```



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;
}
```



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:

I. **Brute Force Method:**
This is the most common method of debugging but is the least efficient method.

II. **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.

III. **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.

IV. **Program Slicing:**
This technique is similar to backtracking. Here the search space is reduced by the 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.

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 Tool is 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.

Dynamic Analysis Tool is a technique that requires 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).

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.

There are four types of integration 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 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.

- **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.

- **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.

- **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 bottom level modules are ready.

Phased VS incremental Testing

- 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 Tests

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

Performance Testing

Performance testing is carried out to check whether the system needs to non-functional requirements identified in the SRS document. There are several types of performance testing:

I. **Stress Testing:** This is also known as ***endurance testing***. This evaluates system performance when it is stressed for short periods of time.

II. **Volume Testing:** It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations.

III. **Configuration Testing:** This is used to analyze system behavior in various hardware and software configurations specified in the requirements.

IV. **Compatibility Testing:** This type of testing is required when the system interfaces with other types of systems.

Error Seeding

Sometimes the customer might specify the maximum number of allowable errors that may be present in the delivered system. These are often expressed in terms of maximum number of allowable errors per line of source code. Error seed can be used to estimate the number of residual errors in a system. Error seeding, as the name implies, seeds the code with some known errors. In other words, some artificial errors are introduced into the program artificially.

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.

SOFTWARE MAINTENANCE

Necessity of Software Maintenance

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immorality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features.

Types of Software Maintenance

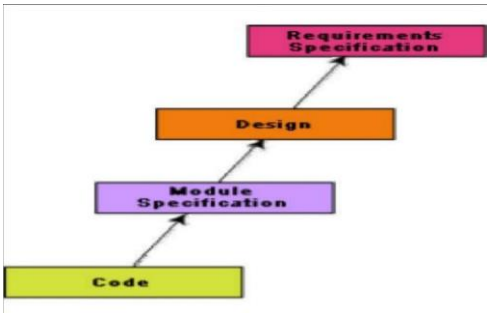
There are basically three types of software maintenance. These are:

- **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

Software Reverse Engineering

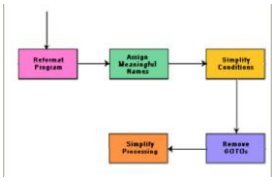
Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis pf its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities. A process model for reverse engineering has been shown in the figure below.



A PROCESS MODEL FOR REVERSE ENGINEERING

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can be beginning. These activities are schematically shown in the figure below. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.



COSMETIC CHANGES CARRIED OUT BEFORE REVERSE ENGINEERING

Legacy Software Products: It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product.