**NAME: EZIMUOBOR GREAT**

**M/N: 15/SCI03/006**

**ASSIGNMENT**

**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:

1. A coding standard gives uniform appearances to the code written by different engineers.

2. It facilitates code of understanding.

3. Promotes good programming practices.

**Characteristics of a Programming Language**

**1. Readability**

**2. Portability**

**3. Generality**

**4. Brevity**

**5. Error checking**

**6. Cost**

**7. Familiar notation**

**8. Modularity**

**9. Wide available**

**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.

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.

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.

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.

**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 or bug). But, the mere presence of an error may not necessarily lead to a failure.

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. 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.

**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.

**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.



Unit testing with the help of driver 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 designor code is required. The following are the two main approaches to designing black box test cases.

i. Equivalence class portioning

ii. Boundary value analysis

 **WHITE-BOX TESTING**

One white-box testing strategy is said to be stronger than another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called complementary. The concepts of stronger and complementary testing are schematically illustrated in fig. 20.1.

 

Fig. 20.1: Stronger and complementary testing strategies

**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 linespreceding this statement that can influence the value of that variable.

**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 immortality 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. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

**Types of software maintenance**

 There are basically three types of software maintenance.

 **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.

**Problems associated with software maintenance**

Software maintenance work typically is much more expensive than what it should be and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work.