NWABINELI EMELDA CHISOM

15/ENG05/016

MECHATRONICS ENGINEERING

ASSIGNMENT 2

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.

Characteristics of a Programming Language

* Readability
* Portability
* Generality
* Brevity
* Error checking
* Cost

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 guidelines recommended by many software development organizations.

1. Do not use a coding style that is too clever or too difficult to understand

2. Avoid obscure side effects

3. Do not use an identifier for multiple purposes

4. The code should be well-documented

5. The length of any function should not exceed 10 source lines

6. Do not use goto statements

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
* Aim of Testing
* Verification Vs Validation
* Design of Test Cases
* Functional Testing Vs. Structural Testing

BLACK-BOX TESTING

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 Partitioning
* 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

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.

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.

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

Brute Force Method

Cause Elimination Method

Program Slicing

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.

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

* Big bang approach
* Bottom- up approach
* Top-down approach
* Mixed-approach

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 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:

1. Alpha Testing.
2. Beta testing.
3. Acceptance Testing

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.

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. 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.

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. During required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

Software Reverse Engineering Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of 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. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well- designed products become legacy software as their structure degrades through a series of maintenance efforts.

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. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.