**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 the design into a code, a good high level language is required.

Programmers adhere to a standard and well defined style of coding which they call their coding standard. The following are some representative coding standards.

* Rules for limiting the use of global.
* Contents of the headers preceding codes for different modules.
* Naming conventions for global variables, local variables and constant identifiers.
* Error return conventions and exception handling mechanisms.

**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. Two types of code review techniques commonly used are code inspection and code walk throughs.

**Code Walk Throughs**

* The main objectives of the walk through are to discover the algorithmic and logical errors in the code.
* A few members of the development team are given the code few days before the walk through meeting to read and understand the code.
* Each member selects some test cases and simulates execution of the code by hand.
* The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.

**Code Inspection**

* The aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.
* Adherence to coding standards is also checked during code inspection.
* Classical programming errors which can be checked during code inspection include jumps into loops, nonterminating loops, incompatible assignments and so on.

**Clean Room Testing**

This type of testing relies heavily on walk throughs, inspection, and formal verification. The software development philosophy is based on avoiding software defects by using a rigorous inspection process. The objective of this software is zero-defect software. The clean room approach to software development is based on five characteristics:

* Formal Specification
* Incremental development
* Structured programming
* Static verification
* Statistical testing of the system

**Software Documentation**

When various kinds of software products are developed, users’ manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed. All these documents enhance understandability and maintainability of a software product, and reduce the effort and time required for maintenance.

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

**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, useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc.

**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.

**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.

**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, an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible must be designed.

**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**

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**

White box testing employs the concepts of stronger and complementary testing. If all types of errors detected by the first testing strategy are also detected by a 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.

DEBUGGING AND INTEGRATION TESTING

Identifying errors in a program code and then fixing them is known as debugging. The following are some of the approaches popularly adopted by programmers for debugging:

* **Brute Force Method:** Loading the program with print statements to print the intermediate values anticipating that some of the printed values will help to identify the statement in error.
* **Backtracking:** 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.
* **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.
* **Program Slicing:** 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.

A program analysis tool is 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. We can classify these into two broad categories: Static Analysis tools and Dynamic Analysis tools.

**Static analysis tools** typically analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold.

**Dynamic analysis** tools usually instrument the code (i.e. add additional statements in the source code to collect program execution traces). The instrumented code when executed allows for recording the behaviour of the software for different test cases. After the software has been tested with its full test suite and its behaviour recorded, the dynamic analysis tool carries 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.

**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 testing approaches:

* **Big-bang approach:** This is the simplest integration testing approach, where all the modules making up a system are integrated in a single step.
* **Bottom-up approach:** In bottom-up testing, each subsystem is tested separately and then the full system is tested. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem.
* **Top-down approach:** 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.
* **Mixed approach:** A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches while overcoming the shortcomings of the top-down and bottom-up approaches. Testing can start as and when modules become available, making this approach the most commonly used integration testing approach.

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

Table 1: Comparison of Integration Testing Strategies

|  |  |
| --- | --- |
| **Phased Integration Testing**  | **Incremental Integration Testing** |
| A group of related modules are added to the partial system each time. | Only one new module is added to the partial system each time. |
| Requires less number of integration steps | Requires more number of integration steps |
| Harder to debug the system with this approach | Easier to debug the system with this approach |

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 refers to the system testing performed by a select group of friendly customers.

**Acceptance Testing:** This refers to the system testing performed by the customer to determine whether he should accept the delivery of the system.

**Error Seeding**

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. The number of these seeded errors detected in the course of the standard testing procedure is determined.

**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**

Software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc. 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.

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. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured.

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. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

After the cosmetic changes have been carried out on legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Fig. 3. 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.

 

Figure 1: Cosmetic changes carried out before reverse engineering

**Legacy software products**

A legacy system is any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, poor structure and lack of technical knowledge. The activities involved in software maintenance depend on several factors such as:

* The extent of modification to the product required
* The resources available to the maintenance team
* The conditions of the existing product
* The expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.