Module
10

Coding and Testing

Lesson
23

Code Review

Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Identify the necessity of coding standards.
- Differentiate between coding standards and coding guidelines.
- State what code review is.
- Explain what clean room testing is.
- Explain the necessity of properly documenting software.
- Differentiate between internal documentation and external documentation.
- Explain what is testing.
- Explain the aim of testing.
- Differentiate between verification and validation.
- Explain why random selection of test cases is not effective.
- Differentiate between functional testing and structural testing.

Coding

Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.

- It enhances code understanding.

- It encourages good programming practices.

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.

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.

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.

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.

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.

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.

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.

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.

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.

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.

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

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 variables, etc.

Clean room testing

Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. 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 name 'clean room' was derived from the analogy with semi-conductor fabrication units. In these units (clean rooms), defects are avoided by manufacturing in ultra-clean atmosphere. In this kind of development, inspections to check the consistency of the components with their specifications has replaced unit-testing.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.

The clean room approach to software development is based on five characteristics:

- **Formal specification:** The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.

- **Incremental development:** The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.

- **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification.

- **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.

- **Statistical testing of the system:** The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on the operational profile which is developed in parallel with the system specification.

The main problem with this approach is that testing effort is increased as walk throughs, inspection, and verification are time-consuming.

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. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.

- Use documents help the users in effectively using the system.

- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.

- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

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

- Internal documentation
- External documentation

Internal documentation 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. Internal documentation is also provided through the 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. Careful experiments

suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without thought. For example, the following style of code commenting does not in any way help in understanding the code.

<div align="center">**a = 10;   /* a made 10 */**</div>

But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.

External documentation 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.

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.

Differentiate between verification and 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..