Module 10

Coding and Testing

Lesson 26

Debugging, Integration and System Testing

Specific Instructional Objectives

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

- Explain why debugging is needed.
- Explain three approaches of debugging.
- Explain three guidelines for effective debugging.
- Explain what is meant by a program analysis tool.
- Explain the functions of a static program analysis tool.
- Explain the functions of a dynamic program analysis tool.
- Explain the type of failures detected by integration testing.
- Identify four types of integration test approaches and explain them.
- Differentiate between phased and incremental testing in the context of integration testing.
- What are three types of system testing? Differentiate among them.
- Identify nine types of performance tests that can be performed to check whether the system meets the non-functional requirements identified in the SRS document.
- Explain what is meant by error seeding.
- Explain what functions are performed by regression 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 lines preceding this statement that can influence the value of that variable [Mund2002].

Debugging guidelines

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 mistakes 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.

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 tools
- Dynamic Analysis tools

Static program analysis tools

Static analysis tool is also 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. The structural properties that are usually analyzed are:

- Whether the coding standards have been adhered to?
- Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walk throughs and code inspections might be considered as static analysis methods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

Dynamic program analysis tools

Dynamic program analysis techniques require 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). The instrumented code when executed allows us to record the behavior of the software for different test cases. After the software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool caries 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. For example, the post execution dynamic analysis report might provide data on extent statement, branch and path coverage achieved.

Normally the dynamic analysis results are reported in the form of a histogram or a pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily and provides evidence that thorough testing has been done. The dynamic analysis results the extent of testing performed in white-box mode. If the testing coverage is not satisfactory more test cases can be designed and added to the test suite. Further, dynamic analysis results can help to eliminate redundant test cases from the test suite.

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

There are four types of integration testing 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 approach
- Top-down approach
- Bottom-up approach
- Mixed-approach

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. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

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. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

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. 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. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

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 top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

Phased vs. incremental testing

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

- 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:

- **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.

- **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.

- **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

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 tests test the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance tests test the conformance of the system with the nonfunctional requirements of the system.

Performance testing

Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document. There are several types of performance testing. Among of them nine types are discussed below. The types of performance testing to be carried out on a system depend on the different non-functional requirements of the system documented in the SRS document. All performance tests can be considered as black-box tests.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing
- Regression testing
- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

Stress Testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multiprogrammed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours. For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

Volume Testing

It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations. For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

Configuration Testing

This is used to analyze system behavior in various hardware and software configurations specified in the requirements. Sometimes systems are built in variable configurations for different users. For instance, we might define a minimal system to serve a single user, and other extension configurations to serve additional users. The system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.

Compatibility Testing

This type of testing is required when the system interfaces with other types of systems. Compatibility aims to check whether the interface functions perform as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression Testing

This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

Recovery Testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as applicable and discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance Testing

This testing addresses the diagnostic programs, and other procedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation Testing

It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance.

Usability Testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, report formats, and other aspects relating to the user interface requirements are tested.

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. The number of these seeded errors detected in the course of the standard testing procedure is determined. These values in conjunction with the number of unseeded errors detected can be used to predict:

- The number of errors remaining in the product.
- The effectiveness of the testing strategy.

Let N be the total number of defects in the system and let n of these defects be found by testing.

Let S be the total number of seeded defects, and let s of these defects be found during testing.

$$n/N = s/S$$

or

$$N = S \times n/s$$

Defects still remaining after testing = $N-n = n\times(S - s)/s$

Error seeding works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. To some extent, the different categories of errors that remain can be estimated to a first approximation by analyzing historical data of similar projects. Due to the shortcoming that the types of seeded errors should match closely with the types of errors actually existing in the code, error seeding is useful only to a moderate extent.

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. The functionality of regression testing has been discussed earlier.

The following questions have been designed to test the objectives identified for this module:

1. What are the different ways of documenting program code? Which of these is usually the most useful while understanding a piece of code?
2. What is a coding standard? Identify the problems that might occur if the engineers of an organization do not adhere to any coding standard.

3. What is the difference between coding standards and coding guidelines? Why are these considered as important in a software development organization?

4. Write down five important coding standards.

5. Write down five important coding guidelines.

6. What do you mean by side effects of a function call? Why are obscure side effects undesirable?

7. What is meant by code review? Why is it required to be completed before performing integration and system testing?

8. Identify the type of errors that can be detected during code walk throughs.

9. Identify the type of errors that can be detected during code inspection.

10. What is clean room testing?

11. Why is it important to properly document a software product?

12. Differentiate between the external and internal documentation of a software product.

13. Identify the necessity of testing of a software product.

14. Distinguish between error and failure. Testing detects which of these two? Justify it.

15. Differentiate between verification and validation in the contest of software testing.

16. Is random selection of test cases effective? Justify.

17. Write down major differences between functional testing and structural testing.

18. Do you agree with the statement: "The effectiveness of a testing suite in detecting errors in a system can be determined by examining the number of test cases in the suite". Justify your answer.

19. What are driver and stub modules in the context of unit testing of a software product?

20. Given a software and its requirements specification document, how can black-box test suites for this software be designed?

21. Identify two guidelines for the design of equivalence classes for a problem.

22. Explain why boundary value analysis is so important for the design of black box test suite for a problem.

23. Compare the features of stronger testing with the features of complementary testing.

24. Which is strongest structural testing technique among statement coverage-based testing, branch coverage-based testing, and condition coverage-based testing? Why?

25. Discuss how does control flow graph (CFG) of a problem help in understanding of path coverage based testing strategy.

26. Draw the control flow graph for the following function named find-maximum. From the control flow graph, determines its Cyclomatic complexity.

```
int find-maximum(int i, int j, int k)
{
        int max;

        if(i>j) then
                if(i>k) then max = i;
                        else max = k;
                else if(j>k) max = j;
                         else max = k;
        return(max);
 }
```

27. What is the difference between path and linearly independent path in terms of control flow graph (CFG) of a problem?

28. Define a metric form which the upper bound for the number of linearly independent paths of a program can be computed.

29. Consider the following C function named bin-search:

```
/* num is the number the function searches in a presorted
integer array arr */

 int bin_search(int num)
 {
        int min, max;
        min = 0;
        max = 100;
        while(min!=max){
        if (arr[(min+max)/2]>num)
                max = (min+max)/2;
        else if(arr[(min+max)/2]<num)
                        min = (min+max)/2;
                else return((min+max)/2); }
        return(-1);
 }
```

Determine the cyclomatic complexity of the above problem.

30. What is meant by data flow-based testing approach?

31. What are the advantages of performing mutation testing upon a software product?

32. Write down three general guidelines for performing effective debugging.

33. Distinguish between the static and dynamic analysis of a program. How are static and dynamic program analysis results useful?

34. What do you understand by the term integration testing? What are the different types of integration testing methods that can be used to carry out integration testing of a large software product?

35. Do you agree with the following statement: "System testing can be considered as a pure black-box test." Justify your answer.

36. What do you understand by performance testing? Write down the different types of performance testing.

37. What is meant by error seeding?

38. Explain the necessity of performing regression testing.

Mark all options which are true.

1. The side effects of a function call include

□ modification of parameters passed by reference
□ modification of global variables
□ modification of I/O operations
□ all of the above

2. Code review for a module is carried out

□ as soon as skeletal code written
□ before the module is successfully compiled
□ after the module is successfully compiled and all the syntax errors have been eliminated
□ before the module is successfully compiled and all the syntax errors have been eliminated

3. An important factor that guides the integration plan for integration testing is

□ ER diagram
□ data flow diagram
□ structure chart
□ none of the above

4. An integration testing approach, where all the modules making up a system are integrated in a single step is known as

□ top-down integration testing
□ bottom-up integration testing
□ big-bang integration testing
□ mixed integration testing

5. An integration testing approach, where testing can start whenever modules become available is known as

□ top-down integration testing
□ bottom-up integration testing
□ big-bang integration testing
□ mixed integration testing

6. When a system interfaces with other types of systems then that time the testing that will be required is

□ volume testing
□ configuration testing
□ compatibility testing
□ maintenance testing

7. When a system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. then the testing required to be performed is:

□ documentation testing
□ regression testing
□ maintenance testing
□ recovery testing

8. Error seed can be used

□ to estimate the total number of defects in the system
□ to estimate the total number of seeded defects in a system
□ to estimate the number of residual errors in a system
□ none of the above

9. Test summary report comprises of

□ the total number of tests that have been applied to a subsystem
□ how many tests have been successful
□ how many tests have been unsuccessful
□ all of the above

Mark the following as either True or False. Justify your answer.

1. Coding standards are synonyms for coding guidelines.
2. During code inspection, you detect errors whereas during code testing you detect failures.
3. Out of all types of internal documentation (i.e. provided in the source code), careful commenting is most useful.
4. Error and failure are synonymous in software testing terminology.
5. Software verification and validation are synonyms terms.
6. The effectiveness of a test suite in detecting errors in a system can be determined by counting the number of test cases in the suite.
7. The number of test cases required for statement coverage-based testing of a program can be greater than those required for path coverage-based testing of the same program.
8. Condition testing strategy is a stronger testing strategy than branch testing strategy.
9. A program can have more than one linearly independent path.
10. Once the McCabe's Cyclomatic complexity of a program has been determined, it is very easy to identify all the linearly independent paths of the program.
11. Introduction of additional edges and nodes in the CFG due to introduction of sequence types of statements in the program can increase the cyclomatic complexity of the program.
12. A pure top-down integration testing does not require the use of any stub modules.
13. Adherence to coding standards is checked during the system testing stage.
14. Development of suitable driver and stub functions are essential for carrying out effective system testing of a product.
15. System testing can be considered as a white box testing.
16. The main purpose of integration testing is to find design errors.