16. What are the advantages of using a window management system for GUI design? Name some commercially available window management systems.
17. Briefly discuss the architecture of the X-window system. What are the important advantages of using the X-window system for developing graphical user interfaces?
18. Write five sentences to highlight the important features of Visual C++.
19. What do you understand by a visual language? How do languages such as Visual C++ and Visual Basic let you do component-based user interface development?
20. What do you understand by component-based user interface development? What are the advantages of component-based user interface development?
21. Why a count of the different screens of the GUI of an application may not be an accurate measure of the size of the user interface? Suggest a more accurate measure of the size of the user interface of an application. Explain how it overcomes the difficulties with the number of screens measure.
22. Distinguish between a "user-centred design" and "design by users". Examine the pros and cons of these two approaches to user interface design.
23. Suppose the customer feedback for a product that you have developed is "too difficult to learn". Explain how the speed of learning of the user interface of the product can be increased.
24. Distinguish between procedural and object-oriented user interface. Which type of interface is more usable? Justify your answer.
25. What do you understand by "look and feel" of a window? Which component of an operating system determines the look and feel of the windows? Explain your answer.
26. Design and develop a graphical user interface for the graphical editor software described in Problem 6.18.
27. Prepare a checklist of at least five inspection items for effective inspection of any user interface.
28. Study the user interface of some popular software products such as Word, PowerPoint, Excel, OpenOffice, Gimp, etc., and identify the metaphors used. Also, examine how tasks are broken up into subtasks and closure achieved.
29. What do you understand by a metaphor in the context of user interface design? Why is a metaphor-based user interface design advantageous? List a few metaphors which can be used for user interface design.
30. What do you understand by a *modal dialog*? Why are these required? Why should the use of too many modal dialogs in an interface design be avoided?
31. How does the human cognition capabilities and limitations influence human-computer user interface designing?
32. Distinguish between a user-centric interface design and interface design by users.
33. Why is "push from below" model of interaction between a GUI object with either a controller or domain object is not a good idea? What solution does MVC pattern offer in this regard?

CHAPTER 10

CODING AND TESTING

In this chapter, we will discuss the coding and testing phases of the software life cycle.

Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.

In the coding phase, every module specified in the design document is coded and unit tested. During unit testing, each module is tested in isolation from other modules. That is, a module is tested independently as and when its coding is complete.

After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.

Integration and testing of modules is carried out according to an integration plan. The integration plan, according to which different modules are integrated together, usually envisages integration of modules through a number of steps. During each integration step, a number of modules are added to the partially integrated system and the resultant system is tested. The full product takes shape only after all the modules have been integrated together. System testing is conducted on the full product. During system testing, the product is tested against its requirements as recorded in the SRS document.

We had already pointed out in Chapter 2 that testing is an important development phase and typically requires the maximum effort among all the development phases. Usually, testing is carried out using a large number of test cases. The different test cases can be executed parallely by different team members. Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed. Thus, the testing phase provides the scope for a large number of parallel activities to take place. In a typical development organization, at any time, the maximum number of software engineers can be found to be engaged in testing activities. Not surprisingly then, in the software industry there is always a large demand for software test engineers. However, many novice engineers bear the wrong impression that testing is a secondary activity and that it is intellectually not as stimulating as the activities associated with the other development phases. As we shall soon realize, testing a software product is as much challenging as initial development activities such as specifications, design, and coding. Moreover, testing involves a lot of creative thinking.

In this chapter, we first take up some important issues associated with the coding phase. Subsequently, we will discuss various types of program testing techniques.

10.1 CODING

The input to the coding phase is the design document produced at the end of the design phase. Please recollect that the design document contains not only the high-level design of the system in the form of a module structure (e.g. a structure chart) but also the detailed design. The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified. During the coding phase, different modules identified in the design document are coded according to their respective module specifications. We can describe the overall objective of the coding phase to be the following:

> 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.

Normally, good software development organizations require their programmers to adhere to some well-defined and standard style of coding which they call their coding standard. These software development organizations formulate their own coding standards that suit them the most, and require their developers to follow the standards rigorously because of the significant business advantages it offers. The main advantages of adhering to a standard style of coding are the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It facilitates code understanding.
- It promotes 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, the error return conventions, etc. Besides the coding standards, several coding guidelines are also prescribed by software companies. But, what is the difference between a coding guideline and a coding standard?

> Coding standards have to be mandatorily followed by the programmers, and compliance to coding standards is verified before the testing phase can start. In contrast, coding guidelines provide some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers.

After a module has been coded, usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing. It is important to detect as many errors as possible during code reviews, because reviews are an efficient way of removing errors from code as compared to defect elimination using testing. We first discuss a few representative coding standards and guidelines. Subsequently, we discuss code review techniques. We then discuss software documentation and various testing techniques in Sections 10.4 and 10.5, respectively.

10.1.1 Coding Standards and Guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what suits their organization best and based on the specific types of

products they develop. To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines which are commonly adopted by many software development organizations, rather than trying to provide an exhaustive list.

Representative coding standards

1. Rules for limiting the use of globals: These rules list what types of data can be declared global and what can not, with a view to limit the data that needs to be defined with global scope.

2. Standard headers to precede the code of 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 is specified. The following is an example of header format adopted some companies:

- Name of the module
- Date on which the module was created
- Author's name
- Modification history
- Synopsis of the module
- Different functions supported in the module, along with their input/output parameters
- Global variables accessed/modified by the module

3. Naming conventions for global variables, local variables, and constant identifiers: A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g. `GlobalData`) and local variable names start with small letters (e.g. `localData`). Constant names should be formed using capital letters only (e.g. `CONSTDATA`).

4. Conventions regarding error return values and exception handling mechanisms. The way error conditions are reported by different functions in a program should be standard within an organization. For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code.

Representative coding guidelines

The following are some representative coding guidelines that are recommended by many software development organizations. Wherever necessary, the rationale behind these guidelines is also mentioned.

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. As a result, it can make maintenance and debugging difficult and expensive.

2. Avoid obscure side effects: The side effects of a function call include modifications to the 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 or some file I/O performed obscurely in a called module, it becomes difficult to infer from the function's name and header information, making it difficult to understand the 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 also computing and storing the final result. The rationale that they give for such multiple use of variables is memory efficiency, e.g. three variables use up three memory locations, whereas when 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 a variable for multiple purposes are 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. For example, while changing the final computed result from integer to float type, the programmer might subsequently notice that it has also been used as a temporary loop variable that can not be a float type.

4. The code should be well-documented: As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.

5. The length of any function should not exceed 10 source lines: A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

6. Do not use GO TO statements: Use of GO TO statements makes a program unstructured, thereby making it very difficult to understand, debug, and maintain the program.

10.2 CODE REVIEW

Code review for a module is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module. Code reviews are extremely cost-effective strategies for eliminating coding errors and for producing high quality code.

> The reason behind why code review is a much more cost-effective strategy to eliminate errors from code compared to testing is that reviews directly detect errors. On the other hand, testing only helps detect failures and significant effort needs to be spent in debugging to locate the error.

The rationale behind the above statement is explained as follows: eliminating an error from code involves three main activities—testing, debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and

under certain circumstances. Once a failure is detected, debugging is carried out to locate the error that is causing the failure. Finally, error correction is carried out to fix the bug. Of the three testing activities, debugging is possibly the most laborious and time-consuming activity. In code inspection, errors are directly detected, circumventing the necessity of debugging.

Normally, the following two types of reviews are carried out on the code of a module:

- Code inspection.
- Code walkthrough.

The procedures for conduction and the final objectives of these two review techniques are very different. In the following two subsections, we discuss these two code review techniques.

10.2.1 Code Walkthrough

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand (i.e. traces the execution through different statements and functions of the code).

The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.

The members note down their findings of their walkthrough and discuss those in a walkthrough meeting where the coder of the module is present.

Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years for making this naive but useful analysis technique more effective. These guidelines are based on personal experience, common sense, several other subjective factors. Therefore, these guidelines should be considered as examples rather than as accepted rules to be applied dogmatically. Some of these guidelines are the following:

- The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among the engineers that they are being evaluated in the code walkthrough meetings, managers should not attend the walkthrough meetings.

10.2.2 Code Inspection

During code inspection, the code is examined for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code

walkthroughs. We can state the principal aim of the code inspection activity to be the following.

> The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer oversights and to check whether coding standards have been adhered to.

As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter. It is more likely that such an error can be discovered by specifically looking for this kinds of mistakes in the code, rather than by simply hand simulating execution of the code. 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 types of errors most frequently committed. Such a list of commonly committed errors can be used as a checklist 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
- Non-terminating 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 equality of floating point values, etc.

10.2.3 Clean Room Testing

Clean room testing was pioneered at IBM. This type of testing relies heavily on walkthroughs, 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.

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 main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time-consuming for detecting all simple errors. Testing-based error correction is efficient for detecting certain errors that escape manual inspection.

10.3 SOFTWARE DOCUMENTATION

When a software product is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc are developed as part of the software engineering process. All these documents are considered a vital part of any good software development practice. Good documents are very useful and serve the following purposes.

- Good documents help enhance understandability of a software product. As a result, the availability of good documents help to reduce the effort and time required for maintenance.
- Documents help the users to understand and effectively use the system.
- Good documents help in effectively tackling the manpower turnover[1] problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
- Production of good documents helps the manager to effectively track the progress of the project. The project manager would know that some measurable progress has been achieved, if the results of some pieces of work has been documented and the same has been reviewed.

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

Internal documentation: These are provided in the source code itself.

External documentation: These are the supporting documents that usually accompany a software product.

We discuss these two types of documentation in the next section.

10.3.1 Internal Documentation

Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following.

- Comments embedded in the source code
- Use of meaningful variable names
- Module and function headers
- Code indentation
- Code structuring (i.e. code decomposed into modules and functions)
- Use of enumerated types
- Use of constant identifiers
- Use of user-defined data types

[1] Manpower turnover is the software industry jargon for denoting the unusually high rate at which personnel attrition occurs (i.e. personnel leave an organization).

Out of those different types of internal documentation, which one is the most valuable?

> Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand a piece of code.

The above assertion, of course, is 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 much thought. For example, the following style of code commenting does not in any way help in understanding the code.

```
a=10; /* a made 10 */
```

A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines. Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding the code.

10.3.2 External Documentation

External documentation is provided through various types of supporting documents such as user manual, software requirements specification document, design document, test document, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

An important feature of good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the product. In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents. Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning's fog index is very useful. We discuss this next.

10.3.3 Gunning's Fog Index

Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, any one who has completed his 12th class would not have much difficulty in understanding that document.

The Gunning's fog index of a document D can be computed as follows:

$$fog(D) = 0.4 \times \left(\frac{words}{sentences}\right) + \%\ of\ words\ having\ 3\ or\ more\ syllables$$

Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the common observation that long sentences are difficult to understand. The second factor measures the percentage of complex words in the document. Note that a syllable is a group of words that can be independently pronounced. For example, the word *sentence* has three syllables (*sen*, *ten*, and *ce*).

An example sentence and its fog index

Consider the following sentence: "The Gunning's fog index is based on the premise that use of short sentences and simple words makes a document easy to understand."

The fog index of the above example sentence is

$$0.4 \times \left(\frac{23}{1}\right) + \left(\frac{4}{23}\right) \times 100 = 26$$

If a users' manual is to be designed for use by factory workers whose educational qualification is class 8, then the document should be written such that the Gunning's fog index of the document does not exceed 8.

10.4 TESTING

The aim of program testing is to identify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose most of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.

10.4.1 Basic Concepts and Terminologies

Testing a program involves 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 input data and the conditions under which the failure occurs are noted for later debugging and error correction. The following are some commonly used terms associated with testing.

- An **error** is a mistake committed by the development team during any of the development phases. The mistake might have been committed in the requirements, design, or code. An error is also sometimes referred to as a fault, a bug, or a defect.
- A **failure** is a manifestation of an **error** (or *defect* or *bug*). In other words, a failure is the symptom of an error. But, mere presence of an error may not necessarily lead to a failure.
- **A test case 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.**

Out of these different types of internal documentation, which one is the most valuable?

> Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand a piece of code

The above assertion, of course, is 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 much thought. For example, the following style of code commenting does not in any way help in understanding the code.

```
a=10; /* a made 10 */
```

A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines. Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding the code.

10.3.2 External Documentation

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

An important feature of good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the product. In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents. Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning's fog index is very useful. We discuss this next.

10.3.3 Gunning's Fog Index

Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, any one who has completed his 12th class would not have much difficulty in understanding that document.

The Gunning's fog index of a document D can be computed as follows:

$$fog(D) = 0.4 \times \left(\frac{words}{sentences}\right) + \%\ of\ words\ having\ 3\ or\ more\ syllables$$

Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the commonsense observation that long sentences are difficult to understand. The second factor measures the percentage of complex words in the document. Note that a syllable is a group of words that can be independently pronounced. For example, the word sentence has three syllables (sen, ten, and ce).

An example sentence and its fog index

Consider the following sentence: "The Gunning's fog index is based on the premise that use of short sentences and simple words makes a document easy to understand."

The fog index of the above example sentence is

$$0.4 * \left(\frac{23}{1}\right) + \left(\frac{4}{23}\right) * 100 = 26$$

If a users' manual is to be designed for use by factory workers whose educational qualification is class 8, then the document should be written such that the Gunning's fog index of the document does not exceed 8.

10.4 TESTING

The aim of program testing is to identify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose most of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.

10.4.1 Basic Concepts and Terminologies

Testing a program involves 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 input data and the conditions under which the failure occurs are noted for later debugging and error correction. The following are some commonly used terms associated with testing.

- **An error** is a mistake committed by the development team during any of the development phases. The mistake might have been committed in the requirements, design, or code. An error is also sometimes referred to as a fault, a bug, or a defect.
- **A failure** is a manifestation of an **error** (or *defect* or *bug*). In other words, a failure is the symptom of an error. But, mere presence of an error may not necessarily lead to a failure.
- **A test case** 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.

- A **test suite** is the set of all test cases with which a given software product is tested.

Verification versus 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. Alternatively, we can state the difference between verification and validation in the following words.

> While verification is concerned with phase containment of errors, the aim of validation is to make the final product error free.

Testing activities

Testing involves performing the following main activities:

1. Test suite design: The set of test cases using which a program is to be tested is designed using different test case design techniques that have been discussed later in this chapter.

2. Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

3. Debugging: For each failure observed during the previous activity, debugging is carried out to identify the statements that are in error. In this, the failure symptoms are analyzed to locate the errors. Debugging techniques are discussed in Section 7.9.

4. Error correction: After the error is located in the previous activity, the code is appropriately changed to correct the error.

Of all the above mentioned testing activities, debugging is possibly the most time consuming activity. The testing activities have been shown schematically in Figure 10.1.

10.4.2 Why Design Test Cases?

Before discussing the various test case design techniques, we need to satisfactorily answer the following questions. Would it not be sufficient to test a software using a large number of random input values? The answer to this question—this would be very costly and at the same time very ineffective way of testing due to the following reasons.

> When test cases are designed based on random input data, many of the test cases do not contribute to the significance of the test suite, That is, they do not help detect any additional defects not already being detected by other test cases in the suite.

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. Let us try to understand why the number of random test cases in a test suite, in general, does not indicate of the effectiveness of testing. Consider the following example code segment which determines the greater of two integer values x and y. This code segment has a

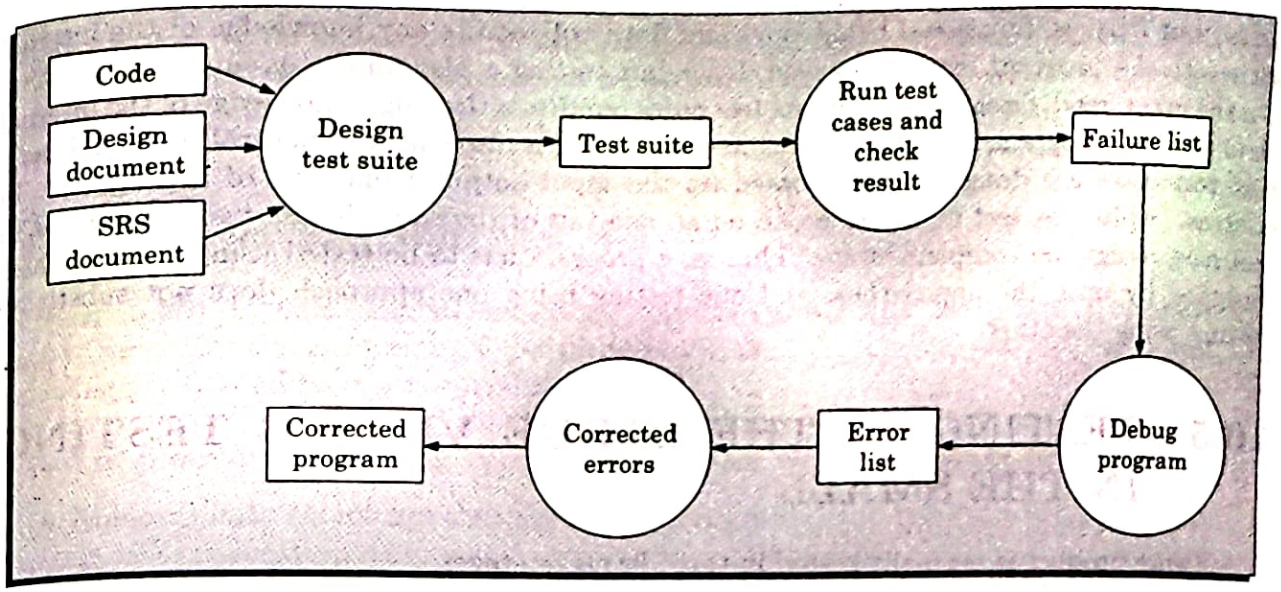


Figure 10.1: The testing process.

simple programming error.

```
if (x>y) max = x;
else max = x;
```

For the given 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. All the test cases in the larger test suite help detect the same error, while the other error in the code remains undetected. 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 for effective testing the test suite should be carefully designed rather than picked randomly.

We have already pointed out that 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, to satisfactorily test a software with minimum cost, we must design a minimal test suite that is of reasonable size and can uncover as many existing errors in the system as possible. To reduce testing cost and at the same time to make testing more effective, systematic approaches have been developed to design a minimal test suite.

A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contrast to testing using some random input values.

There are essentially two main approaches to systematically design test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/out

behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as functional testing. On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of a program, and therefore white-box testing is also called structural testing. Black-box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code. These two approaches to test case design are complementary. That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

10.5 TESTING IN THE LARGE VERSUS TESTING IN THE SMALL

A software product is normally tested in three levels or stages:

- Unit testing
- Integration testing
- System testing

During unit testing, the individual components (or units) of a program are tested.

Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.

After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing). Finally, the fully integrated system is tested (system testing). Integration and system testing are known as **testing in the large**.

Often beginners ask a question—why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly? The answer to this question is, there are two main reasons to it. First, while testing a module, other modules with which this module needs to interface may not be ready. Moreover, it is always a good idea to first test the module in isolation before integration because it makes debugging easier. If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

In the following, we discuss the different levels of testing. It should be borne in mind in all our subsequent discussions that unit testing is carried out in the coding phase itself as soon as coding of a module is complete. On the other hand, integration and system testing are carried out during the testing phase.

10.6 UNIT TESTING

Unit testing is undertaken after a module has been coded and reviewed. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit

under test has to be developed. In this section, we first discuss the environment needed to perform unit testing. In the subsequent subsections, we discuss the different approaches to design test cases for unit testing.

10.6.1 Driver and Stub Modules

In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module:

- The procedures belonging to other modules that the module under test calls.
- Nonlocal data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

Stub

The role of stub and driver modules is pictorially shown in Figure 10.2. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure, but has a highly simplified behaviour. For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism.

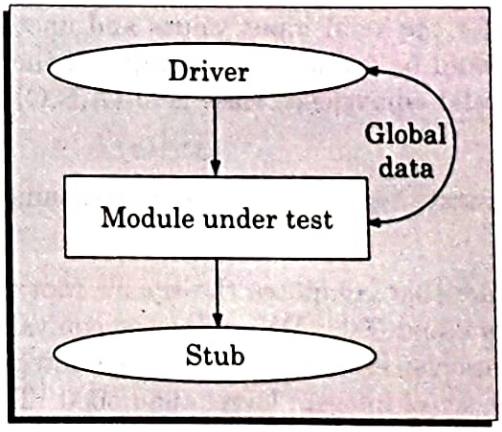


Figure 10.2: Unit testing with the help of driver and stub modules.

Driver

A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the module under test with appropriate parameter values for testing.

10.7 BLACK-BOX TESTING

In 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 available to design black-box test cases:

- Equivalence class partitioning
- Boundary value analysis

In the following, we elaborate these two test case design techniques.

10.7.1 Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

> The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.

Equivalence classes for a program can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e. [1,10]), then the invalid equivalence classes are $[-\infty, 0], [11, +\infty]$.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is ∪-{A,B,C}, where ∪ is the universe of possible input values.

In the following, we illustrate equivalence class partitioning-based test case generation through four examples.

Example 10.1: For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence class test suite.

There are three equivalence classes—the set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {−5, 500, 6000}.

Example 10.2: Design the equivalence class test cases for a program that reads two integer pairs (m_1, c_1) and (m_2, c_2) defining two straight lines of the form `y=mx+c`. The program computes the intersection point of the two straight lines and displays the point of intersection.

The equivalence classes are the following:

- Parallel lines ($m_1 = m_2, c_1 \neq c_2$)
- Intersecting lines ($m_1 \neq m_2$)
- Coincident lines ($m_1 = m_2, c_1 = c_2$)

Now, selecting one representative value from each equivalence class, we get the required equivalence class test suite {(2, 2)(2, 5), (5, 5)(7, 7), (10, 10)(10, 10)}.

Example 10.3: Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

The equivalence classes have been shown in Figure 10.3. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc, aba, abcdef}.

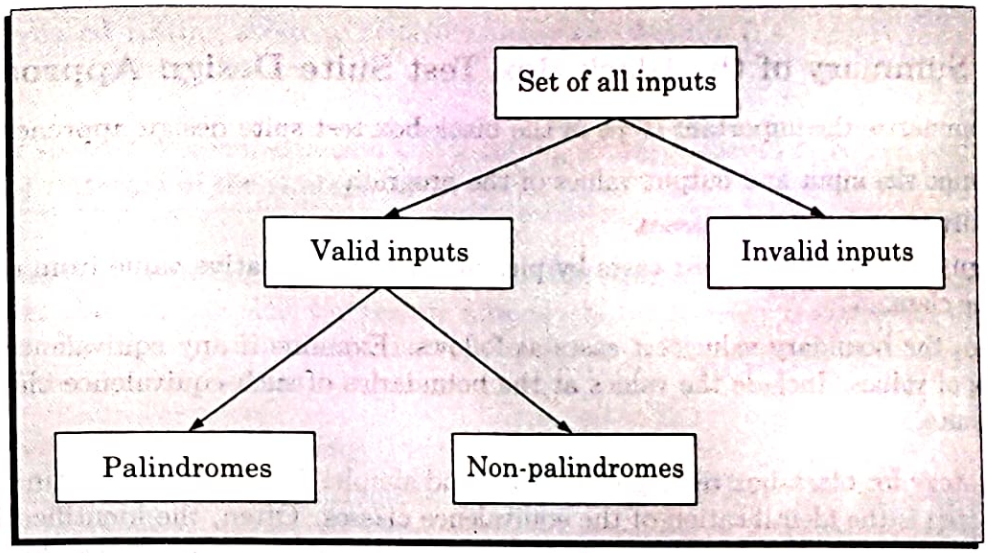


Figure 10.3: Equivalence classes for Example 10.3.

10.7.2 Boundary Value Analysis

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes

that are not a range of values (that is, consist of a discrete collection of values) no boundary value test cases can be defined. For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.

Example 10.4: For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

There are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1,5000,5001}.

Example 10.5: Design boundary value test suite for the function described in Example 10.3.

The equivalence classes have been showed in Figure 10.3. There is a boundary between the valid and invalid equivalence classes. Thus, the boundary value test suite is {abc, abcdef}.

10.7.3 Summary of the Black-Box Test Suite Design Approach

We now summarize the important steps in the black-box test suite design approach:

- Examine the input and output values of the program.
- Identify the equivalence classes.
- Design equivalence class test cases by picking one representative value from each equivalence class.
- Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

The strategy for black-box testing is intuitive and simple. For black-box testing, the most important step is the identification of the equivalence classes. Often, the identification of the equivalence classes is not straightforward. However, with little practice we would be able to identify all equivalence classes in the input data domain. Without practice, you may overlook many equivalence classes in the input data set. Once the equivalence classes are identified, the equivalence class and boundary value test cases can be selected almost mechanically.

10.8 WHITE-BOX TESTING

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

10.8.1 Basic Concepts

A white-box testing strategy can either be coverage-based or fault-based.

Fault-based testing

A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the fault model of the strategy. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

Coverage-based testing

A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

A coverage-based testing strategy typically targets to execute (i.e. cover) certain program elements for discovering failures.

> The set of specific program elements that a testing strategy requires to be executed is called the *testing criterion* of the strategy.

For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is *statement coverage*. We can, in other words, say that tests that are adequate with respect to a criterion, cover all elements of the domain defined by that criterion.

Stronger versus weaker testing

We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.

> A white-box testing strategy is said to be stronger than another strategy, if all types of program elements covered by the second testing strategy are also covered by the first testing strategy, and the first strategy additionally covers some more types of elements not covered by the second strategy.

When none of two testing strategies fully covers the program elements exercised by the other, then the two are called **complementary testing strategies**. The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.4. Observe in Figure 10.4(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.4(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.

> If a stronger testing has been performed, then a weaker testing need not be carried out.

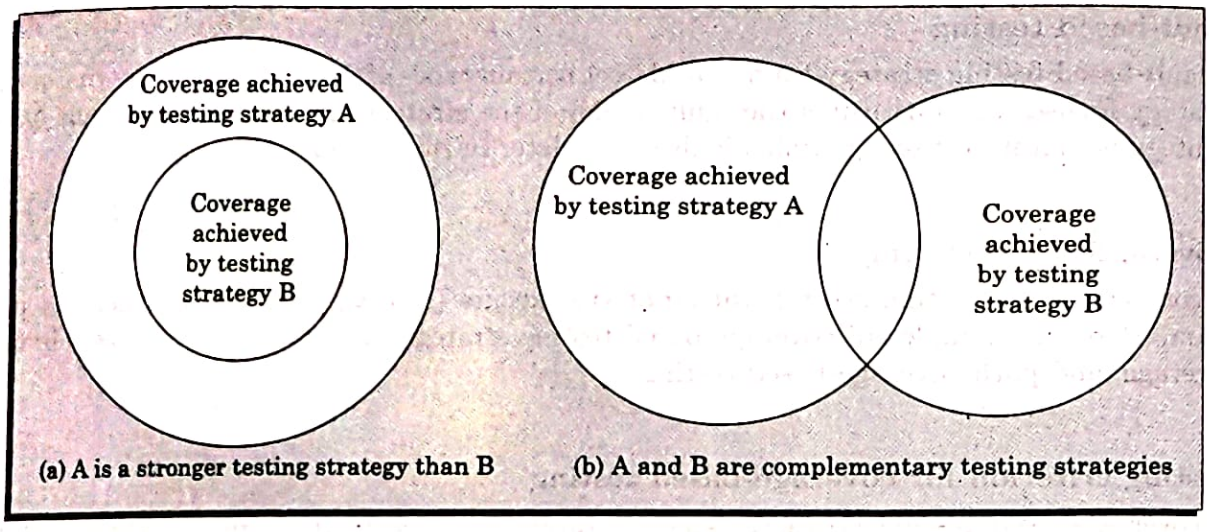


Figure 10.4: Illustration of stronger, weaker, and complementary testing strategies.

A test suite should, however, be enriched by using various complementary testing strategies.

10.8.2 Statement Coverage

The statement coverage-based strategy aims to design test cases so as to execute every statement in a program at least once.

> The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.

It is obvious that without executing a statement, it is difficult to determine whether it leads to a failure due to some illegal memory access, wrong result computation due to improper arithmetic operation, etc. It can however be pointed out that a weakness of the statement-coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values. Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate how test cases can be designed using the statement coverage strategy.

Example 10.6: Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
    int x,y;
{
     1 while (x != y){
     2    if (x>y) then
     3          x=x-y;
     4    else y=y-x;
     5 }
```

```
        6 return x;
}
```

To design the test cases for the statement coverage, the conditional expression of the `while` statement needs to be made true and the conditional expression of the `if` statement needs to be made both true and false. By choosing the test set {(x=3,y=3),(x=4,y=3), (x=3,y=4)}, all statements of the program would be executed at least once.

10.8.3 Branch Coverage

Branch coverage-based testing requires test cases to be designed so as to make each branch condition in the program to assume true and false values in turn. Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once. For the program of example 1, the test cases for branch coverage can be {(x=3,y=3),(x=3,y=2), (x=4,y=3), (x=3,y=4)}.

It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

Theorem 10.8.1: Branch coverage-based testing is stronger than statement coverage-based testing.

Proof: We need to show that (i) branch coverage ensures statement coverage, and (ii) statement coverage does not ensure branch coverage.

(i) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).

(ii) To show that statement coverage does not ensure branch coverage, it would be sufficient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.

```
if(x>2) x+=1;
```

The test suite would achieve statement coverage. However, it should not achieve branch coverage, since the condition $(x > 2)$ would not be made false.

10.8.4 Condition Coverage

In the condition coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression $((c_1.and.c_2).or.c_3)$, the components c_1, c_2 *and* c_3 are each made to assume both true and false values. Branch testing is probably a simplified condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2^n test cases are required for condition coverage. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

ath Coverage

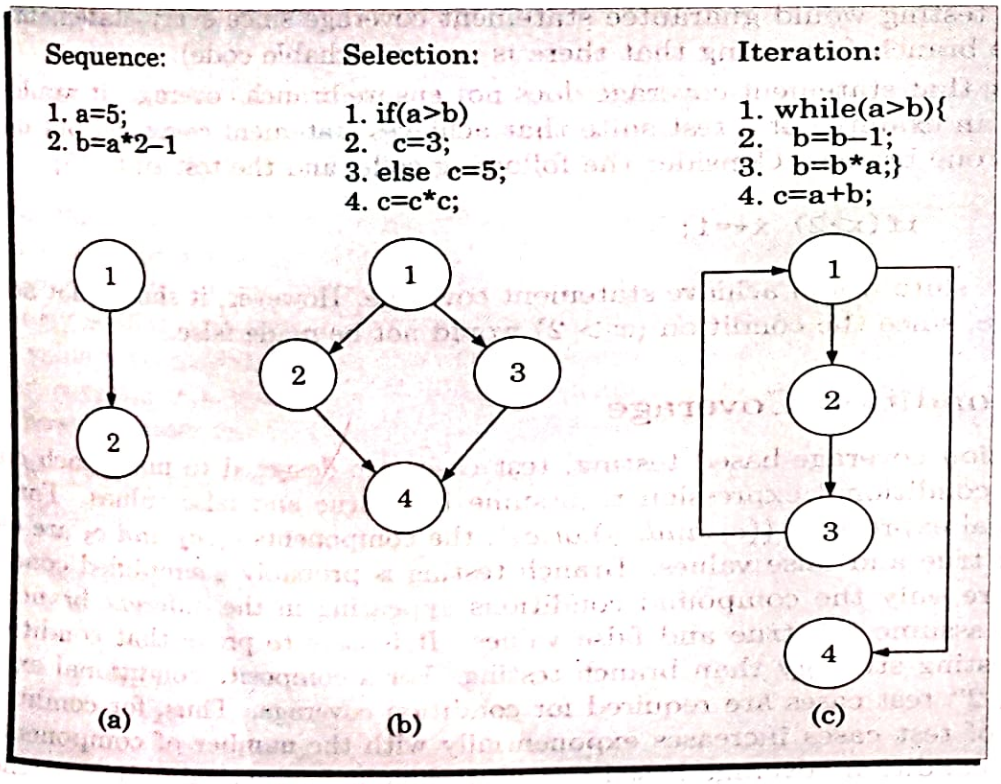
e-based testing strategy requires designing test cases such that all linearly inde-
s (or basis paths) in the program are executed at least once. A linearly indepen-
n be defined in terms of the control flow graph (CFG) of a program. Therefore,
nderstand the path coverage-based testing strategy, we need to first understand
G of a program can be drawn.

ow Graph (CFG)

w graph describes how the control flows through the program. In other words, we
control flow graph as the following:

low graph describes the sequence in which the different instructions of a program
ed.

to draw the control flow graph of a program, we need to first number all the
of a program. The different numbered statements serve as nodes of the control
see Figure 10.5). There exists an edge from one node to another, if the execution
ment representing the first node can result in the transfer of control to the other



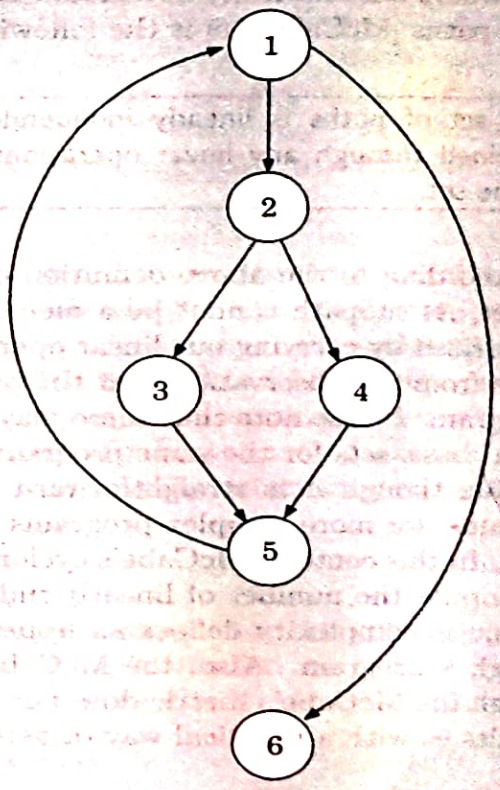
e 10.5: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a of nodes and edges (N, E), such that each node $n \in N$ corresponds to a unique program ement, and an edge exists between two nodes if control can transfer from one node to the er.

We can easily draw the CFG for any program, if we know how to represent the sequence, ction, and iteration types of statements in the CFG. After all, every program is constructed n these three types of constructs only. Figure 10.5 summarizes how the CFG for these three es of constructs can be drawn. The CFG representation of the sequence and decision types tatements is straightforward. Please note carefully how the CFG for the loop (iteration) struct can be drawn. For iteration type of constructs such as the while construct, the loop dition is tested only at the beginning of the loop, and therefore, always control flows from last statement of the loop to the top of the loop. That is, the loop construct terminates n the first statement (after the loop is found to be false) and does not at any time exit the at the last statement of the loop. Using these basic ideas, the CFG of the program given igure 10.6(a) can be drawn as shown in Figure 10.6(b).

```
int compute_gcd(int x, int y) {
  1 while(x!=y) {
  2     if(x>y) then
  3       x=x-y;
  4     else y=y-x;
  5   }
  6   return x;
    }
```

(a) An example program



(b) Control flow graph

Figure 10.6: Control flow diagram of an example program.

th

path through a program is any node and edge sequence from the start node to a terminal le of the control flow graph of a program in the presence of return or exit types of statements.

Please note that a program can have more than one terminal nodes when it contains multiple `exit` or `return` type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops. For example, in Figure 10.5(c), there can be an infinite number of paths such as 12314, 12312314, 12312312314, etc. If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage-based testing has been designed, to not to cover all paths, but only a subset of paths called linearly independent paths (or basis paths). Let us now discuss what are linearly independent paths and how to identify them in a program.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called **linearly independent set of paths** (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths. This is because, any path having a new node would automatically have a new edge. An alternative definition of a linearly independent set of paths [McCabe, 76] is the following.

> If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e. additions or subtractions) on the other paths in the set.

According to the above definition of a linearly independent set of paths, for any path in the set, its subpath cannot be a member of the set. In fact, any path of a program, can be synthesized by carrying out linear operations on the basis paths. Possibly, the name basis set comes from the observation that the paths in the basis set form the basis for all the paths of a program. Please note that there may not always exist a unique basis set for a program and several basis sets for the same program can usually be defined.

Even though it is straightforward to identify the linearly independent paths for simple programs, for more complex programs it is not easy to determine the number of independent paths. In this context, McCabe's cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Though the McCabe's metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.

10.8.6 McCabe's Cyclomatic Complexity Metric

McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

Method 1

Given a control flow graph G of a program, the cyclomatic complexity $V(G)$ can be computed as:

$$V(G) = E - N + 2$$

where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in Figure 10.6, $E = 7$ and $N = 6$. Therefore, the value of cyclomatic complexity is

$$7 - 6 + 2 = 3$$

Method 2

An alternate way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

```
V(G) = Total number of non-overlapping bounded areas + 1
```

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a **bounded area**. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar (i.e. however, you draw the graph, two or more edges always intersect). Actually, it can be shown that control flow representation of structured programs always yields planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity would not give the correct answer.

The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown in Figure 10.6, from a visual examination of the CFG the number of bounded areas is 2. Therefore, the cyclomatic complexity, computed with this method is also $2 + 1 = 3$. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the method for computing CFGs can easily be automated. That is, it can be easily coded into a program that can be used to automatically determine the cyclomatic complexities of arbitrary CFGs.

Method 3

The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe's metric is equal to $N + 1$.

How is path testing carried out based on computed McCabe's cyclomatic metric value?

Knowing the number of basis paths in a program does not make it any easier to design the test cases, only it gives an indication of the minimum number of test cases required for path coverage. For the CFG of a moderately complex program segment of say 20 nodes and 25 edges, you may need several days of effort to identify all the linearly independent paths in it and to design the test cases. It is therefore impractical to require the test designers to identify all the linearly independent paths in a code, and then design the test cases to force

execution along each of the identified paths. In practice, for path testing, usually the tester keeps on forming test cases with random data and executes those until the required coverage is achieved. A testing tool such as a dynamic program analyzer (see section 10.10.2) is used to determine the percentage of linearly independent paths covered by the test cases that have been executed so far. If the percentage of linearly independent paths covered is below 90%, more test cases (with random inputs) are added to increase the path coverage. Normally, it is not practical to target achievement of 100% path coverage, since, the McCabe's metric is only an upper bound and does not give the exact number of paths.

Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program.

1. Draw control flow graph.
2. Determine $V(G)$.
3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
4. Repeat
   Test using a randomly designed set of test cases.
   Perform dynamic analysis to check the path coverage achieved.
   until at least 90 per cent path coverage is achieved.

Other uses of McCabe's cyclomatic complexity metric

Beside its use in path testing, cyclomatic complexity of programs has many other interesting applications as follows:

1. Estimation of structural complexity of code: McCabe's cyclomatic complexity is a measure of the **structural complexity** of a program. The reason for this is that it is computed based on the code structure (number of decision and iteration constructs used). This is in contrast to the computational complexity that is based on the execution of the program statements. Intuitively, the McCabe's complexity metric correlates with the difficulty level of understanding a program, since one understands a program by understanding the computations carried out along all independent paths of the program.

Cyclomatic complexity of a program is a measure of the psychological complexity or the level of difficulty in understanding the program.

In view of the above result, from the maintenance perspective, it makes good sense to limit the cyclomatic complexity of the different modules to some reasonable value. Good software development organizations usually restrict the cyclomatic complexity of different functions to a maximum value of ten or so.

2. Estimation of testing effort: Cyclomatic complexity is a measure of the maximum number of basis paths. Thus, it indicates the minimum number of test cases required to achieve path coverage. Therefore, the testing effort and the time required to test a piece of code satisfactorily is proportional to the cyclomatic complexity of the code. To reduce testing effort, it is necessary to restrict the cyclomatic complexity of every function to seven.

3. **Estimation of program reliability:** Experimental studies indicate there exists a clear relationship between the McCabe's metric and the number of errors latent in the code after testing. This relationship exists possibly due to the correlation of cyclomatic complexity with the structural complexity of code. Usually the larger is the structural complexity, the more difficult it is to test and debug the code.

10.8.7 Data Flow-based Testing

Data flow-based testing method selects test paths of a program according to the definitions and uses of different variables in a program.

Consider a program P. For a statement numbered S of P, let

$$\text{DEF(S)} = \frac{X}{\text{Statement } S \text{ contains a definition of } X}$$

$$\text{USES(S)} = \frac{X}{\text{Statement } S \text{ contains a use of } X}$$

For the statement `S: a=b+c;`, DEF(S)={a}, USES(S)={b,c}. The definition of variable X at statement S is said to be live at statement S_1, if there exists a path from statement S to statement S_1 which does not contain any definition of X.

The definition-use chain (or DU chain) of a variable X is of the form $[X, S, S_1]$, where S and S_1 are statement numbers, such that

$$X \in DEF(S)$$

and

$$X \in USES(S_1)$$

And the definition of X in the statement S is live at statement S_1. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are especially useful for testing programs containing nested if and loop statements.

10.8.8 Mutation Testing

All white-box testing strategies that we have discussed so far, are coverage-based testing techniques. In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, the software is first tested by using an initial test suite designed by using various white-box testing strategies that we have discussed. After the initial testing is complete, mutation testing can be taken up.

The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a **mutated program** and the change effected is called a **mutant**. Depending on the change made to the code, a mutated program may or may not introduce some errors. A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement.

A mutated program is tested against the original test suite of the program. If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has

successfully been detected by the test suite. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant.

An important advantage of mutation testing is that it can be automated to a great extent. The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be simple program alterations such as deleting a statement, deleting a variable definition, changing the type of an arithmetic operator (e.g. + to -), changing a logical operator (and to or) changing the value of a constant, changing the data type of a variable, etc. A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Since mutation testing requires generating a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would automatically generate the mutants and run all the test cases automatically. At present, several test tools are available that automatically generate mutants for a given program.

10.9 DEBUGGING

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed. In this section, we shall summarize the important approaches that are available to identify the error locations. Each of these approaches has its own advantages and disadvantages, and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

10.9.1 Debugging Approaches

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

Brute force method

This is the most common method of debugging, but is the least efficient method. In this approach, print statements are inserted through out the program 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 for complex programs, limiting the use of this approach.

source files preceding this statement that can influence the value of that variable [Prasad 2002].

10.9.2 Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity and experience. 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 program design may require an inordinate amount of effort to be put into debugging even for 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 (see Section 10.14) must be carried out.

10.10 PROGRAM ANALYSIS TOOLS

A program analysis tool usually is an automated tool that takes either 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, adequacy of testing, etc. We can classify various program analysis tools into the following two broad categories:

- Static analysis tools
- Dynamic analysis tools

These two categories of program analysis tools are discussed in the following.

10.10.1 Static Analysis Tools

Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyze the source code to compute certain metrics characterizing the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions, such as whether some properties such as the following hold:

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

Code review techniques such as code walkthrough and code inspection discussed in sections 10.2.1 and 10.2.2 might be considered as static analysis methods since those target to detect errors based on analyzing the source code. However, strictly speaking, this is not true since we are using the term static program analysis to denote automated analysis tools. On the other hand, a compiler can be considered to be a type of a static program analysis tool.

A major practical limitation of the static analysis tools lies in their inability to analyze run-time information such as dynamic memory references (pointer variables and pointer arithmetic, etc.). In high-level programming languages, pointer variables and dynamic memory allocation provide the capability for dynamic memory references. Dynamic memory referencing is a major source of programming errors in a program. Static analyzers cannot evaluate pointer values and array subscripts.

Static analysis tools often summarize the results of analysis of every function in a polar chart known as **Kiviat chart**. A Kiviat chart typically shows the analyzed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead's metrics, etc.

10.10.2 Dynamic Analysis Tools

Dynamic program analysis tools can be used to evaluate several program characteristics based on analysis of the run-time behaviour of a program. These tools usually record and analyze the actual behaviour of a program while it is being executed. A dynamic program analysis tool (also called a dynamic analyzer) usually collects execution trace information by instrumenting the code. The code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program. The instrumented code when executed, records the behaviour of the software for different test cases.

An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.

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 coverage that has been achieved by the complete test suite for the program. For example, the dynamic analysis tool can report the statement, branch, and path coverage achieved by a test suite. If the coverage achieved is not satisfactory more test cases can be designed, added to the test suite, and run. Further, dynamic analysis results can help eliminate redundant test cases from a test suite.

Normally the dynamic analysis results are reported in the form of a histogram or 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 to provide evidence that thorough testing has been carried out.

10.11 INTEGRATION TESTING

Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module. Thus, the primary objective of integration testing is to test the module interfaces, i.e. there are no errors in parameter passing, when one module invokes the functionality of another module.

> The objective of integration testing is to check whether the different modules of a program interface with each other properly.

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. We have already discussed in Chapter 6 that a structure chart (or module dependency graph) specifies the order in which different modules call each other. Thus, by examining the structure chart, the integration plan can be developed. Any one (or a mixture) of the following approaches can be used to develop the test plan:

- Big bang approach
- Top-down approach
- Bottom-up approach
- Mixed (also called sandwiched) approach

In the following, we provide an overview of these approaches to integration testing.

Big-bang integration testing

Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply linked together and tested. However, this technique can meaningfully be used 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 lie in any of the modules. Therefore, debugging errors reported during big bang integration testing are very expensive to fix. As a result, big-bang integration testing is almost never used for large programs.

Bottom-up integration testing

Large software products are often made up of several subsystems. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated. Thus, the subsytems can be integrated separately and independently.

The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems. The principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

Top-down integration testing

Top-down integration testing starts with the root module and one or two subordinate modules in the system. After the top-level skeleton has been tested, the modules that are at the immediately lower layer 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. An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers. 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 the input output (I/O) operations.

Mixed integration testing

The 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 approach, testing can start as and when modules become available after unit testing. Therefore, this is one of the most commonly used integration testing approaches. In this approach, both stubs and drivers are required to be designed.

10.11.1 Phased versus Incremental Integration Testing

Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partially integrated system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Obviously, 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 while using the incremental testing approach since the errors can easily be traced to the interface of the recently integrated module. Please observe that a degenerate case of the phased integration testing approach is big-bang testing.

10.12 TESTING OBJECT-ORIENTED PROGRAMS

During the initial years of object-oriented programming, it was believed that object-orientation would, to a great extent, reduce the cost and effort incurred on testing. This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc. thereby minimizing the chances of errors in the code. However, it was soon realized that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs. The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that are present in procedural programs. Therefore, additional test cases are needed to be designed to detect these. We examine these issues as well as some other basic issues in testing object-oriented programs in the following.

10.12.1 What is a Suitable Unit for Testing Object-Oriented Programs?

For procedural programs, we had seen that procedures are the basic units of testing. That is, first all the procedures are unit tested. Then various tested procedures are integrated together and tested. Thus, as far as procedural programs are concerned, procedures are the basic units of testing. Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing? Weyuker studied this issue and postulated his anticomposition axiom as follows.

> Adequate testing of individual methods does not ensure that a class has been satisfactorily tested.

The main intuitive justification for the anticomposition axiom is the following. A method operates in the scope of the data and other methods of its object. Therefore, it is necessary to test a method in the context of these. Moreover, objects can have significant states. The behaviour of a method can be different based on the state of the corresponding object. Therefore, it is not enough to test all the methods and check whether they can be integrated satisfactorily. A method has to be tested with all the other methods and data of the corresponding object. Moreover, a method needs to be tested at all the states that the object can assume. As a result, it is improper to consider a method as the basic unit of testing an object-oriented program.

> An object as the basic unit of testing of object-oriented programs.

Thus, in an object-oriented program, unit testing would mean testing each object in isolation. During integration testing (called cluster testing in the object-oriented testing literature) various unit tested objects are integrated and tested. Finally, system-level testing is carried out.

10.12.2 Do Various Object-Orientation Concepts Make Testing Easy?

In this section, we discuss the implications of different object-orientation concepts in testing.

Encapsulation

We had discussed in Chapter 7 that the encapsulation feature helps in data abstraction, error isolation, and error prevention. However, as far as testing is concerned, encapsulation leads to difficulty in testing and debugging. Encapsulation prevents the tester from accessing the data internal to an object. Of course, it is possible that one can require classes to support state reporting methods to print out all the data internal to an object. Thus, the encapsulation feature though makes testing difficult, the difficulty can be overcome to some extent through use of appropriate state reporting methods.

Inheritance

The inheritance feature helps in code reuse and was expected to simplify testing. It was expected that if a class is tested thoroughly, then the classes that are derived from this class would need only incremental testing of the added features. However, this is not the case.

> Even if the base class has been thoroughly tested, the methods inherited from the base class need to be tested again in the derived class.

The reason for this is that the inherited methods would work in a new context (new data and method definitions). As a result, correct behaviour of a method at an upper level, does not guarantee correct behaviour at a lower level. Therefore, retesting of inherited methods needs to be followed as a rule, rather as an exception.

Dynamic binding

Dynamic binding was introduced to make the code compact, elegant, and easily extensible. However, as far as testing is concerned all possible bindings of a method call have to be identified and tested. This is not easy since the bindings take place at run-time.

Object states

In contrast to the procedures in a procedural program, objects store data permanently. As a result, objects do have significant states. The behaviour of an object is usually different in different states. That is, some methods may not be active in some of its states. Also, a method may act differently in different states. For example, when a book has been issued out in a library information system, the book reaches the `issuedOut` state. In this state, if the issue method is invoked, then it may not exhibit its normal behaviour.

In view of the discussions above, testing an object at one of its state is not enough. The object has to be tested at all its possible states. Also, whether all the transitions between states

(as specified in the object model) function properly or not should be tested. Additionally, it needs to be tested that no extra (sneak) transitions exist, neither are there (extra) states present other than those defined in the state model. For state-based testing, it is therefore helpful to have the state model of the objects, so that the conformance of the object to its state model can be tested.

10.12.3 Why are Traditional Techniques Considered Unsatisfactory for Testing Object-Oriented Programs?

We have already seen that in traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs. Besides this, there are many other significant differences as well between testing procedural and object-oriented programs. For example, statement coverage-based testing which is popular for testing procedural programs is not meaningful for object-oriented programs. The reason is that inherited methods have to be retested in the derived class. In fact, the different object-oriented features (inheritance, polymorphism, dynamic binding, state-based behaviour, etc.) require special test cases to be designed compared to the traditional testing as discussed in section 10.12.2. The various object-orientation features are explicit in the design models, and it is usually difficult to extract from and analysis of the source code. As a result, the design model is a valuable artifact for testing object-oriented programs. Test cases designed based on the design model. Therefore, this approach is considered to be intermediate between a fully white-box and a fully black-box approach, and is called a grey-box approach. It needs to be remembered that, grey-box testing is important for object-oriented programs. This is in contrast to testing procedural programs.

10.12.4 Grey-Box Testing of Object-Oriented Programs

As we have already mentioned, model-based testing is important for object-oriented programs, as these test cases help detect specific to the object-orientation constructs.

For object-oriented programs, several types of test cases can be designed based on the design model. These are called the grey-box test cases.

The following are some important types grey-box of testing that can be carried on based on UML models:

State-model-based testing

1. **State coverage:** Each method of an object are tested at each state of the object.
2. **State transition coverage:** It is tested whether all transitions depicted in the state model work satisfactorily.
3. **State transition path coverage:** All transition paths in the state model are tested.

Use case-based testing

1. **Scenario coverage:** Each use case typically consists of a mainline scenario and several

alternate scenarios. For each use case, the mainline and alternate sequences are covered to check if any errors show up.

Class diagram-based testing

1. Testing derived classes: All derived classes of the base class have to be instantiated and tested. Inherited methods must be retested.

2. Association testing: All association relations are tested.

3. Aggregation testing: Various aggregate objects are created and tested.

Sequence diagram-based testing

1. Method coverage: All methods depicted in the sequence diagrams are covered.

2. Message path coverage: All message paths that can be constructed from the sequence diagrams are covered.

10.12.5 Integration Testing of Object-Oriented Programs

There are two main approaches to integration testing of object-oriented programs:

- Thread-based
- Use based

Thread-based approach

In this approach, all classes that need to collaborate to realize the behaviour of a single use case are integrated and tested. After all the required classes for a complete use case are integrated and tested, another use case is taken up and other classes (if any) necessary for the second use case to run are integrated and tested. This is continued till all use cases have been considered.

Use-based approach

Use-based integration begins by testing classes that either need no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

10.13 SYSTEM TESTING

After all the units of a program have been integrated together and tested, system testing is taken up.

System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation (procedural or object-oriented) is immaterial.

There are essentially three main kinds of system testing:

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

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

3. **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, the test cases can be the same, but the difference is with respect to who designs test cases and carries out testing.

The system test cases can be classified into functionality and performance test cases.

The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests, on the other hand, test the conformance of the system with the non-functional requirements of the system. We have already discussed how to design the functionality test cases by using a black-box approach (in Section 10.7 in the context of unit testing). So, in the following subsection we shall only discuss performance testing.

10.13.1 Performance Testing

Performance testing is an important type of system testing.

Performance testing is carried out to check whether the system meets the non-functional requirements identified in the SRS document.

There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document. All performance tests can be considered as black-box tests.

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 concurrent transactions, then the system is stressed by attempting to initiate 15 or more transactions 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 their maximum capacity, but may be 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 stress testing the response time is checked with exactly 60 users working simultaneously.

Volume testing

Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle 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

Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements. Sometimes systems are built to work in different configurations for different users. For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing. The system is configured in each of the required configurations and depending on the specific customer requirements, 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 external systems (e.g. databases, servers, etc.). 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 also discussed in Section 10.14.

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

Documentation testing

It is checked whether 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, messages, report

formats, and other aspects relating to the user interface requirements are tested. However, a GUI being just being functionally correct is not enough. The GUI has to be checked against the checklist we discussed in section 9.5.6.

Security testing

Security testing is essential for software products that process confidential data. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers.

10.13.2 Error Seeding

Sometimes customers specify the maximum number of residual errors that can be present in the delivered software. These requirements are often expressed in terms of maximum number of allowable errors per line of source code. The error seeding technique can be used to estimate the number of residual errors in a software.

Error seeding, as the name implies, seeds the code with some known errors. In other words, some artificial errors are introduced (seeded) into the program. The number of these seeded errors that are detected in the course of standard testing is determined. These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program:

- 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. Therefore, we get

$$n/N = s/S$$

or

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

Defects still remaining after testing is

$$N - n = n \times ((S - 1)/s)$$

Error seeding works satisfactorily only if the kind 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 are latent can be estimated by analyzing historical data collected from similar projects. That is, the data collected is regarding the types and the frequency of latent errors for all earlier related projects. This gives an indication of the types (and the frequency) of errors that are likely to have been committed in the program under consideration. Based on these data, the different types of errors with the required frequency of occurrence can be seeded.