APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

KTU ASSIST

a complete app for ktu students

Get it on Google Play

www.ktuassist.in

Module 4

Coding and Testing

CODING

Ques 1) What is coding?

Ans: Coding

Coding is an activity that succeeds software design. In coding, the design is converted to runnable code. The purpose of coding activity is to convert a specification into a representation understandable by the machine. The aim is to implement the design in the best possible manner.

The coding phase affects both testing and maintenance phases that follow. The time spent in coding is a small percentage of the total software cost, while testing and maintenance consume the major percentage. The goal during coding should not be to reduce the implementation cost, but the goal should be to reduce the cost of later phases. Thus, the goal is to write source code and internal documentation so that conformance of the code to its specifications can be easily verified, and so that debugging, testing and modification are eased. This goal can be achieved by making the source code as clear and straightforward as possible.

The hallmarks of good programs include:

1) Simplicity
2) Clarity
3) Elegance

Whereas obscurity, cleverness and complexity are indications of inadequate design and misdirected thinking.

Source code clarity is enhanced by structured coding techniques, by good coding style, by appropriate supporting documents, by good internal comments, and by the features provided in modern programming languages.

Coding can be subject to companywide standards that may define the entire layout of programs, such as the headers for comments in every unit, naming conventions for variables, classes, and functions, the maximum number of lines in each component, and other aspects that the company deems worthy of standardization.

Implementation team should be provided with a well-defined set of software requirements, an architectural design specification, and a detailed design description. Also each member must understand the objectives of implementation.

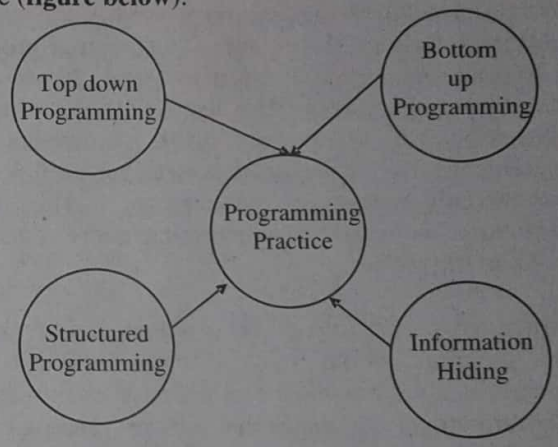
Ques 2) Explain the programming practice in detail?

Ans: Programming Practice

Programming refers to the method of creating a sequence of instructions to enable the computer to perform a task. It is done by developing logic and then writing instructions in a programming language. A program can be written using various programming practices available.

A programming practice refers to the way of writing a program and is used along with coding style guidelines.

Some of the commonly used programming practices include (**figure below**):



1) **Top-Down Programming:** Top-down programming focuses on the use of modules. It is therefore also known as modular programming. The program is broken up into small modules so that it is easy to trace a particular segment of code in the software program. The modules at the top level are those that perform general tasks and proceed to other modules to perform a particular task. Each module is based on the functionality of its functions and procedures.

   In this approach, programming begins from the top level of hierarchy and progresses towards the lower levels. The implementation of modules starts with the main module. After the implementation of the main module, the subordinate modules are implemented and the process follows in this way.

   In top-down programming, there is a risk of implementing data structures as the modules are dependent on each other and they have to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested levels of commands.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

2) **Bottom-Up Programming:** Bottom-up programming refers to the style of programming where an application is constructed with the description of modules. The description begins at the bottom of the hierarchy of modules and progresses through higher levels until it reaches the top. Bottom-up programming is just the opposite of top-down programming. Here, the program modules are more general and reusable than top-down programming.

It is easier to construct functions in bottom-up manner. This is because bottom-up programming requires a way of passing complicated arguments between functions. It takes the form of constructing abstract data types in languages such as C++ or Java, which can be used to implement an entire class of applications and not only the one that is to be written. It therefore becomes easier to add new features in a bottom-up approach than in a top-down programming approach.

3) **Structured Programming:** Structured programming is concerned with the structures used in a computer program. Generally, structures of computer program comprise decisions, sequences, and loops. The decision structures are used for conditional execution of statements (**for example,** 'if' statement). The sequence structures are used for the sequentially executed statements. The loop structures are used for performing some repetitive tasks in the program.

Structured programming forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Some languages such as Ada, Pascal, and dBase are designed with features that implement the logical program structure in the software code.

Primarily, the structured programming focuses on reducing the following statements from the program.

i) 'GOTO' statements.
ii) 'Break' or 'Continue' outside the loops.
iii) Multiple exit points to a function, procedure, or subroutine. **For example,** multiple 'Return' statements should not be used.
iv) Multiple entry points to a function, procedure, or a subroutine.

Structured programming generally makes use of top-down design because program structure is divided into separate subsections. A defined function or set of similar functions is kept separately. Due to this separation of functions, they are easily loaded in the memory. In addition, these functions can be reused in one or more programs. Each module is tested individually. After testing, they are integrated with other modules to achieve an overall program structure. A key characteristic of a structured statement is the presence of single entry and single exit point. This characteristic implies that during execution, a structured statement starts from one defined point and terminates at another defined point.

4) **Information Hiding:** Information hiding focuses on hiding the non-essential details of functions and code in a program so that they are inaccessible to other components of the software. A software developer applies information hiding in software design and coding to hide unnecessary details from the rest of the program.

The objective of information hiding is to minimise complexities among different modules of the software. The complexities arise when one program or module in software is dependent on several other programs and modules.

Information hiding is implemented with the help of interfaces. An interface is a medium of interaction for software components that are using the properties of the software modules containing data. The implementation of interfaces depends on the syntax and process. **Examples** of interface include constants, data types, types of procedures, and so on. Interfaces protect other parts of programs when a software design is changed.

Generally, the interfaces act as a foundation to modular programming (top-down programming) and object-oriented programming. In object-oriented programming, interface of an object comprises a set of methods, which are used to interact with the objects of the software programs. Using information hiding, a single program is divided into several modules. These modules are independent of each other and can be used interchangeably in other software programs.

For example, let us consider a program written for 'car'. The program can be organized in several ways. One is to arrange modules without using information hiding. In this case, the modules can be created as 'front part', 'middle part', and 'rear part'. On the other hand, creating modules using information hiding includes specifying names of modules such as 'engine' and 'steering'.

On comparison, it is found that modules created without using information hiding affect other modules. This is because when a module is modified, it affects the data, which does not require modification. However, if modules are created using information hiding, then modules are concerned only with specific segments of the program and not the whole program or other parts of the program. In **example**, this statement means that the module 'engine' does not have any affect on the module 'steering'.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Ques 3) What is programming style? What is the importance of programming style?

Or

What are the characteristics of good programming style?

Ans: Programming Style

Programmers have their own style of writing programs. Good programming style makes the code easy to understand, not only for other people but also the programmer himself. The program itself help the reader to understand what it does quickly because only a small fraction of the developers if any, are maintaining the program they wrote. Others will, and they must be able to understand what the program does.

Bad programming style makes program difficult to understand, hard to modify, and impossible to maintain over a long period of time, even by the person who coded it originally.

Importance of Programming Style

Programming style is important for the following main reasons:

1) **To Simplify Software Maintenance**
   i) Readability implies clarity and understandability
   ii) Programs have to be read to be maintained
2) **To Avoid Problems**
   i) Good style can avoid common bugs and typos
   ii) Good style can make mistakes clear
3) **To Achieve Readability**
   i) By developers and users
   ii) Now and over years to come

Characteristics of Good Programming Style

A good programming style is characterised by the following:

1) Simplicity,
2) Readability,
3) Good documentation,
4) Changeability,
5) Predictability
6) Consistency in input and output,
7) Module independence, and
8) Good structure

Ques 4) What are the general programming style rules.

Ans: General Programming Style Rules

Some of the rules that are applied when coding using good programming style:

1) **Names:** Selecting module and variable names is, often, not considered important by novice programmers. Only when one starts reading programs written by others, where the variable names are cryptic and not representative, does one realize the importance of selecting proper names. Most variables in a program reflect some entity in the problem domain, and the modules reflect some process.
2) Variable names should be closely related to the entity they represent, and module names should reflect their activity. It is bad practice to choose cryptic names just to avoid typing or totally unrelated names. It is also bad practice to use the same name for multiple purposes. Few **example** names are Student, Employee, inum, input ( ), getStatus ( ).
3) **Control constructs:** It is desirable that as much as possible single entry, single-exit constructs be used. It is also desirable to use a few standard control constructs rather than using a wide variety of constructs, just because they are available in the language, i.e., sequence, selection and iteration constructs which are single-entry, single-exit.
4) **Gotos:** Gotos should be used sparingly and in a disciplined manner. Only, when the alternative to using gotos is more complex, should the gotos be used. In any case, alternatives must be thought of, before finally using a goto. If a goto must be used, forward transfers (or a jump to a later statement) are more acceptable than a backward jump. Use of gotos for exiting a loop or for invoking error handlers is quite acceptable (many languages provide separate constructs for these situations, in which case those constructs should be used). Often, we do not use goto statement to write structured programs as structured programming is called 'Goto Less' programming.
5) **Information hiding:** As discussed earlier, information hiding should be supported where possible. Only the access functions for the data structures should be made visible while hiding the data structure behind these functions.
6) **User-defined types:** Modern languages allow users to define types like the enumerated type. When such facilities are available, they should be exploited where applicable. **For example,** when working with dates, a type can be defined for the day of the week:

   In C enum day = {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

   In Pascal type days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

   Variables can then be declared of this type. Using such types makes the program much clearer than defining codes for each day and then working with codes.
7) **Nesting:** The different control constructs, particularly the if-then-else, can be nested. If the nesting becomes too deep, the programs become harder to understand. In case of deeply nested if-then-else, it is often difficult to determine the **if** statement to which a particular **else** clause is associated. Where possible, deep nesting should be avoided, even if it means a little inefficiency. For example, consider the following construct of nested if-then-else:

```
If C1 then S1
    else if C2 then S2
    else if C3 then S3
    else if C4 then S4;
```

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

If the different conditions are disjoint (as they often are), this structure can be converted into the following structures:

If C1 then S1;
If C2 then S2;
If C3 then S3;
If C4 then S4;

This sequence of statements will produce the same result as the earlier sequence (if the conditions are disjoint), but it is much easier to understand. The price is a little inefficiency in that the latter conditions will be evaluated even if a condition evaluates to true, while in the previous case, the condition evaluation stops when one evaluates to true. Other such situations can be constructed, where alternative program segments can be constructed, to avoid a deep level of nesting. In general, if the price is only a little inefficiency, it is more desirable to avoid deep nesting.

8) **Module Size:** A programmer should carefully examine any routine with very few statements (say fewer than 5) or with too many statements (say more than 50). Large modules often will not be functionally cohesive, and too-small modules might incur unnecessary overhead. There can be no hard-and-fast rule about module sizes; the guiding principle should be cohesion and coupling.

9) **Module Interface:** A module with a complex interface should be carefully examined. Such modules might not be functionally cohesive and, might be implementing multiple functions. As a rule of thumb, any module whose interface has more than five parameters should be carefully examined and broken into multiple modules with a simpler interface, if possible.

10) **Program Layout:** How the program is organized and presented can have great effect on the readability of it. Proper indentation, blank spaces, and parentheses should be used to enhance the readability of programs. Automated tools are available to "pretty print" a program, but it is good practice to have a clear layout of programs.

11) **Side Effects:** When a module is invoked it sometimes has side effects of modifying the program state, beyond the modification of parameters listed in the module interface definition, for example, modifying global variables. Such side effects should be avoided where possible, and if a module has side effects, they should be properly documented.

12) **Robustness:** A program is robust if it does something planned even for exceptional conditions. A program might encounter exceptional conditions is such forms as incorrect input, the incorrect value of some variable, and overflow. A program should try to handle such situations. In general, a program should check for validity of inputs, where possible, and should check for possible overflow of the data structures. If such situations do arise, the program should not just "crash" or "core dump"; it should produce some meaningful message and exit gracefully.

Ques 5) Discuss the verification with respect to coding?

Or

Explain how to verify the program? What are the different methods of program verification?

Ans: Verification

The goal of verification is to assess and improve the quality of the work products generated during development and modification of software.

As per Boehm, Verification is: "Are we building the product right?

Verification is used in order to verify other aspects of the system, for example, performance. This aspect is particularly important in real-time systems where performance may determine whether the system is correct or not.

Program verification falls into two categories:

1) Static Techniques/Methods: Static techniques/ methods do not involve actual program execution on actual numeric data, though it may involve some form of conceptual execution. In static techniques, the program is not compiled and then executed, as in testing.

   Common forms of static techniques are code reading, code reviews and walkthroughs, and symbolic execution. In static techniques, often the errors are detected directly, unlike dynamic techniques where only the presence of an error is detected. This aspect of static testing makes it quite attractive and economical.

2) **Dynamic Methods:** In dynamic method, the program is executed on some test data and the outputs of the program are examined to determine if there are any errors present. Hence, dynamic techniques follow the traditional pattern of testing, and the common notion of testing refers to this technique.

It has been found that the types of errors detected by the two categories of verification techniques are different. The type of errors detected by static techniques is often not found by testing, or it may be more cost-effective to detect these errors by static methods. Consequently, testing and static methods are complimentary in nature and both should be used for reliable software.

Ques 6) Describe the complexity analysis.

Or

Explain the following terms:

1) **Size Measure**
2) **Halstead Measure**
3) **Live Variables**
4) **Knot Count**
5) **Topological Complexity**

Ans: Complexity Analysis

The productivity, in terms of lines of code per unit time, can vary a lot depending on the complexity of the system to be developed. Clearly, a programmer will produce a lesser amount of code for highly complex system programs, as compared to a simple application program.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Similarly, complexity has great impact on the cost of maintaining a program. To quantify complexity with which a program is constructed, some metrics are required to measure.

Cyclomatic complexity is one such complexity measure in which the complexity of a module is the number of independent cycles in the flow graph of the module. Few of the complexity measures are discussed as under:

1) **Size Measure:** A complexity measure tries to capture the level of difficulty in understanding a module, i.e., it tries to quantify a cognitive aspect of a program. It is well known that, in general, the larger a module, the more difficult it is to comprehend. Hence, the size of a module can be taken as a simple measure of the complexity of the module. It can be seen that, on an average, as the size of the module increases, the number of decisions in it are likely to increase. This means that, on an average, as the size increases the cyclomatic complexity also increases.

2) **Extension to Cyclomatic Complexity:** Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

   McCabe has proposed this measure, which considers each decision in the program as a single unit (in the flow graph, one node is created for each condition). With this, the cyclomatic complexity is the same as (under some conditions) the number of decisions plus one.

   Myers has extended the notion of cyclomatic complexity noting that decisions with multiple conditions (combined into a predicate by Boolean operators) are typically more complex than decisions with one condition. In this extension, complexity is measured as an interval rather than a single value. The lower bound of the interval is the number of decisions plus one or the cyclomatic complexity of the program. The upper bound of the interval is the number of individual conditions plus one. Complexity as an interval accounts for both the decisions and the conditions in a program.

   Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways:

   i) The number of regions corresponds to the cyclomatic complexity.

   ii) Cyclomatic complexity, V(G), for a flow graph, G, is defined as
   $V(G) = E - N + 2$

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

   iii) Cyclomatic complexity, V(G), for a flow graph, G, is also defined as
   $V(G) = P + I$

   where P is the number of predicate nodes contained in the flow graph G.

3) **Halstead Measure:** Halstead's complexity measure focused on the internal complexity of a module, as does McCabe's complexity measure. Thus the complexity of the module's connection with its environment is not given much importance. In Halstead's measure, a module's connection with its environment is reflected in terms of operands and operators. A call to another module is considered an operator, and all the parameters are considered operands of this operator.

   Various variables used are n_1 (number of unique operators), and n_2 (number of unique operands), N_1 (total frequency of operators), and N_2 (total frequency of operands). As any program must have at least two operators – one for function call and one for end of statement – the ratio $n_1/2$ can be considered the relative level of difficulty due to the larger number of operators in the program. The ration N_2/n_2 represents the average number of times an operand is used. In a program in which variables are changed more frequently, this ratio will be larger. As such programs are harder to understand, ease of reading or writing is defined as:

   $$D = \frac{n_1 * N_2}{2 * n_2}$$

   In Halstead measure, a module's connection with its environment is reflected in terms of operands and operators.

4) **Live Variables:** To construct or understand a statement, a programmer must keep track of a number of variables, other than those directly used in the statement. For a statement, such data items are called live variables. Intuitively, the more live variables for statements, the harder it will be to understand a program. Hence, the concept of live variables can be used as a metric for program complexity.

   A variable is considered live from its first to its last reference within a module, including all statements between the first and last statement where the variable is referenced. Using this definition, the set of live variables for each statement can be computed easily by analysis of the module's code. The procedure of determining the live variables can easily be automated.

   The number of live variables represents the degree of difficulty of the statement. This notion can be extended to the entire module by defining the average number of live variables. The average number of live variables is the sum of the count of live variables (for all executable statements) divided by the number of executable statements. This is a complexity measure for the module.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

5) **Knot Count:** A programmer, to understand a given program, typically draws allows from the point of control transfer to its destination, helping to create a mental picture of the program and the control transfers in it. According to this metric, the more intertwined these arrows become, the more complex the program. This notion is captured in the concept of a "knot".

Problems can arise while determining the knot count of programs using structured constructs. One method is to convert such a program into one that explicitly shows control transfers and then compute the knot count. The basic scheme can be generalised to flow graphs, though with flow graphs only bounds can be obtained.

6) **Topological Complexity:** Topological complexity is based on the flow graph of a module or program like cyclomatic complexity. The complexity of a program is considered its maximal intersect number min.

To compute the maximal intersect, a flow graph is converted into a strongly connected graph (by drawing an arrow from the terminal node to the initial node). A strongly connected graph divides the graphs into a finite number of regions. The number of regions is (edges – nodes + 2). If we draw a line that enters each region exactly once, then the number of times this line intersects the arcs in the graphs is the maximal intersect min, which is taken to be the complexity of the program.

Ques 7) Discuss the coding standards and guidelines?

Or

Write short note on the Coding standards?

Ans: Coding Standards and Guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their needs and the types of products they develop.

Following are the **Representative Coding Standards:**

1) **Rules for Limiting Use of Global:** These rules list what types of data can be declared global and what cannot.

2) **Contents of Headers Preceding Codes for Different Modules:** The information contained in the headers of different modules should be standard for an organisation. The exact format in which the header information is organized can also be specified. The following are some standard header data:
   i) Name of the module,
   ii) Date on which the module was created,
   iii) Author's name,
   iv) Modification history,
   v) Synopsis of the module,
   vi) Different functions supported, along with their input/output parameters,
   vii) Global variables accessed/modified by the module.

3) **Naming Conventions for Global Variables, Local Variables, and Constant Identifiers:** A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always in capital letters.

4) **Error Return Conventions and Exception Handling Mechanisms:** The way error conditions are reported by different functions in a program and the way common exception conditions are handled, should be standard within an organization.

Following are the **Representative Coding Guidelines:**

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. It also makes maintenance difficult.

2) **Avoid Obscure Side Effects:** The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code.

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 computing and storing the final result. The rationale that is usually given by these programmers for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. Some of the problems caused by use of variables for multiple purposes are as follows:
   i) 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.
   ii) Use of variables for multiple purposes usually makes future enhancements more difficult.

4) **Code Should be Well-Documented:** As a rule of thumb, there must be atleast one comment line on the average for every three source lines.

5) **Length of Any Function should not exceed Ten Source Lines:** A function that is very lengthy is usually very difficult to understand as it probably carries-out many different functions. For the same reason, lengthy functions are likely to have disproportionately a larger number of bugs.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

6) **Do not Use GOTO Statements:** Use of GOTO statements make a program unstructured and very difficult to understand.

TESTING

Ques 8) What is software testing?

Ans: Software Testing

The method of testing the software product is known as 'software testing'. If the software testing is done effectively then it will provide more reliable and accurate results, decrease maintenance cost and also increases the user's satisfaction. So, software products will be of enhanced quality.

One of the critical elements of Software Quality Assurance (SQA) is software testing. It denotes the final review of specification, design and the code that has been generated.

Software testing must be carried out in parallel with the development process. It helps in validation and verification of the software product.

The software testing will provide information to stakeholders regarding the quality of service or product.

According to IEEE, "software testing is the process of analysing a software item to detect the differences between existing and required conditions (i.e., bugs) and to evaluate the features of the software item".

Ques 9) Discuss the fundamental terms used in software testing.

Or

What is error, fault, test case and test suite?

Ans: Testing Fundamentals

While testing the software product, a few testing fundamentals must be needed which play a significant role during the testing. Some of the terms that are common during the software testing phase are as follows:

1) **Errors:** Error is defined as the deviation of the observed, computed or measured output from the specified, true or theoretically correct output. Thus, the difference between the desired or correct output and the output that is actually obtained from the software is known as 'error'. Error is usually defined as the mismatch between the specification of the program and the output obtained in reality.

   According to the IEEE, "Error is the human mistake that caused fault."

   Types of Errors

   There are mainly two types of errors:

   i) **Syntax Error:** When error occurs in the source code of the program, it is known as 'syntax error'. Whenever the statements of the program do not conform to the syntax of the language that is being used for coding, a syntax error will occur as it is necessary that computer program must follow the syntax strictly so that successful compilation takes place.

   ii) **Logic Error:** When the data fields are wrong, invalid combinations are provided or the terms are out of bound then logic error takes place. A logic error is also known as 'bug' due to which the program functions wrongly, but not to halt abnormally. Undesired or unintended outputs or behaviours are resulted due to the logical errors. These errors do not come into light instantly.

2) **Fault:** When a system is unable to perform the desired functions then it is known as 'fault'. This is one of the main reasons that lead to malfunctioning of the software. The main reason behind unintended and unexpected behaviour of a program is that incorrect data definition or step or processes have been carried out.

   According to IEEE, "Fault is the discrepancy in code that causes a failure".

   When the programmers write the code for the software then faults come into it. Faults may also get introduced into the software when it is being designed or when new features are being added to it. New faults might also come in when the identified faults are being corrected.

3) **Failure:** If a component or system fails to carry out a task as per the specification it is known as a 'failure'. If the actual behaviour of the software is different from the specified behaviour, it gives rise to a failure. Both performance and functional causes may lead to failure.

   According to IEEE, "Failure occurs when the external behaviour is incorrect".

4) **Test Case:** A software testing document, in which input, output, action, event and actual and expected results are defined, is known as a 'test case' or 'test case document'.

   According to IEEE 829-1998, "A test case is an input in an expected result."

   A test case comprise of a number of instructions that have been designed with an aim of uncovering a specific defect or error in the software system.

   An input description and the associated desired output description are mentioned in the test case. It is defined as a triplet [I, S, O], where 'I' is the data input fed to the system, 'S' is the relevant system state to which the data is fed, and 'O' is the output expected from the system.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

If the test case is well designed, the probability of finding the error increases. Identification of good test case is the major objective of the test case designer.

5) **Test Suite:** Test suite is made up from a group of test cases that have been combined and related with a group of scenarios.

All the test cases which will be tested using the given software make up a test suite. It may comprise of all the possible test cases. A test suite can be generated by taking any combination of the test cases as it targets to comprise every good and effective test case.

A test suite is also known as 'validation suite'. It is a group of test cases that can be used to show whether some specified behaviours exist in software or not.

A test suite also comprises of elaborated goals or instructions for every test case. While performing the testing, the information regarding the configuration of the system can also be used. Detailed description about the tests that programmer follows and preceding steps and states are also given in the test cases. There are also some incorrect terms for test cases such as 'test scenario', 'test script' or 'test plan'.

The scope or cycle is taken into consideration while creating the test suites. It can consist of two types of tests such as functional and non-functional tests.

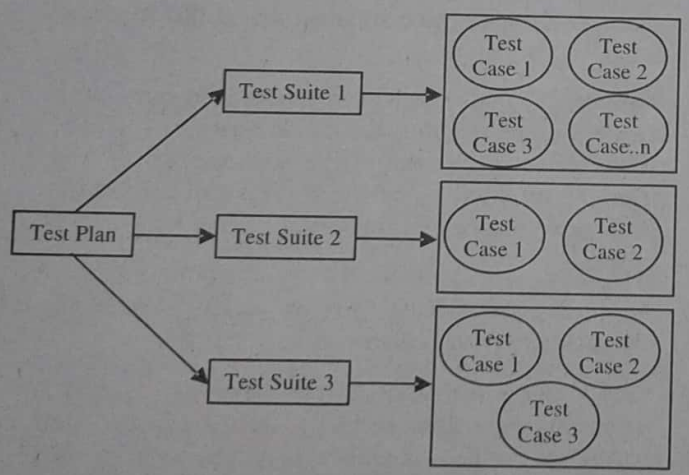


Figure 4.1: Test Suite

Ques 10) What are the objectives of software testing?

Ans: Testing Objectives

Testing performs a significant role in the software development process. Testing is integral part of software development lifecycle. Major significance of software testing lies in its objectives. Software testing is usually performed for the following objectives:

1) **Software Quality Improvement:** Software quality means the conformance to the specified software design requirements. Being correct, the minimum requirement of quality, means performing as required under specified circumstances.

2) **Verification and Validation (V&V):** Another important objective of software testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work.

3) **Software Reliability Estimation:** Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subject to.

The objective of testing is to discover the residual design errors before delivery to the customer. The failure data during the testing process are taken down in order to estimate the software reliability. The testing process may function with regular feedback from the reliability analysis to the testers and designers that is shown in **figure 4.2**:

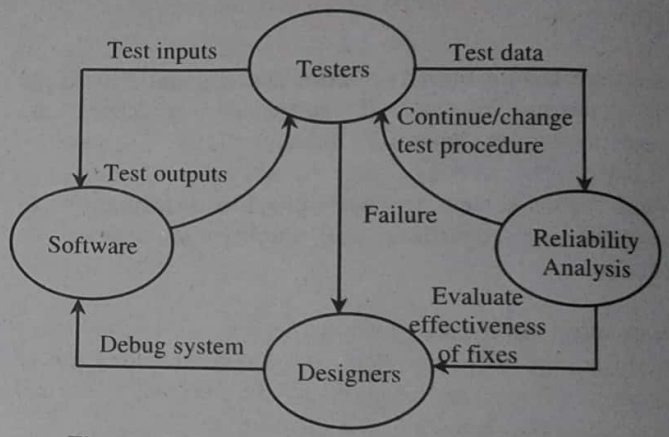


Figure 4.2: Role of Reliability in Software Testing

Based on an operational profile, testing can serve as a statistical sampling method to gain failure data for software reliability estimation.

Ques 11) What are the principles of software testing?

Ans: Software Testing Principles

Testing of software is considered to be not only intellectually challenging but also extremely creative. **According to Davis**, the following testing principles exist:

1) **All Tests should be Traceable to Customer Requirements:** This is carried out to discover those defects that may lead to failure of the system or program. This is a prime criterion of meeting the requirements of the client.

2) **Tests should be Planned Long before Testing Begins:** It is necessary to begin the test planning as soon as the requirements model is done. After designing the design model, the test cases can be planned in detail.

3) **Pareto Principle Applies to Software Testing:** According to the 80:20 rule proposed by **Pareto**, it has been seen that 80% of errors can be traced by just 20% of the total program components.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

4) **Testing should Begin "in the Small" and Progress toward Testing "in the Large":** The individual components are targeted and focussed on during the initial testing. In the advanced stages, if the components are tested in clusters, eventually the whole system is tested as an integration of the components.

5) **Exhaustive Testing is not Possible:** The execution of every path is not possible at the time of testing.

6) **To be most Effective, Testing should be conducted by an Independent Third Party:** The developer of the system is not the right person to carry out every test for the system.

Ques 12) What is testability? What are the characteristics of testability?

Ans: Testability

Testability defines the ease with which tests can be carried out on a program. It can be also defined as the degree to which the test criteria are established in the program. It also determines the extent to which the criteria have been met after executing the tests.

According to James Bach, "Software testability is simply how easily a computer program can be tested".

Characteristics of Testability

Various characteristics of testability are as follows:

1) **Easy to Operate:** The testing of quality software can be done in a better way. If attributes such as quality are given importance while designing and implementing the software, chances of detecting errors during testing can be reduced significantly.

2) **Observability:** Testers have the ability to trace the accuracy of an output for a given input by merely observing it.

3) **Decomposability:** It is easy to test the modules and isolate the problems, if the software can be broken down into independent modules.

4) **Stability:** Software is said to be stable if existing tests can be performed on it and the changes incorporated into the software can be easily controlled.

5) **Easy to Understand:** Software can be tested efficiently if it is easily understandable. Maximum information must be gathered about software so that it can be easily understood. **For example**, detailed information about the code of the software can be gathered by looking at its documentation. This eases the process of testing.

Ques 13) Discuss about the software testing process.

Ans: Software Testing Process

Figure 4.3 shows the model of testing process, as used in plan driven development:

1) **Test cases** are specifications of the inputs to the test and the output from the system (the test results), plus a statement of what is being tested.

2) **Test data** are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible, as people who understand what the system is supposed to do must be involved to specify the expected test results.

3) **Test execution** can be automated.

4) The **expected results** are automatically compared with the predicted results so there is no need for a person to look for errors and anomalies in the test run.

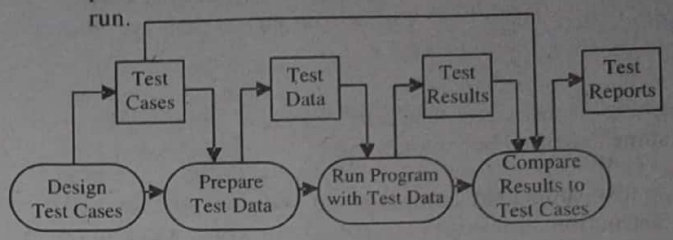


Figure 4.3: Software Testing Process

Ques 14) What are the different software testing techniques?

Or

What is white box and black box testing?

Or

What is Structural and Functional testing?

Ans: Software Testing Techniques

Before the final delivery of the system is given to the user, testing must be carried out in proper manner on the developed software. There are two techniques that provide systematic guidance to the process of test designing (**figure 4.5**).

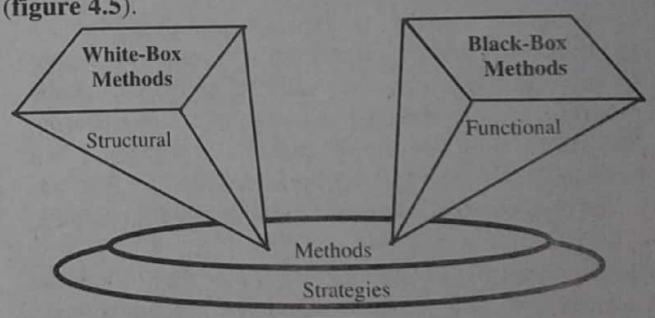


Figure 4.5: Testing Techniques

Following are these techniques:

1) **White-Box Testing/Structural Testing**: The actual code of the program or module that is to be tested is taken into consideration at the time of generating the test cases when this testing is used.

   The implementation of the program is the prime concern of structural testing. In this testing, the various data structures and different programming used in the program are exercised, not the various input and output conditions though they are the by-products.

   In structural testing, the test cases are generated in such a manner that every desired structure of the program is covered. This is why this testing is sometimes also known as **coverage-based testing**.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Statement coverage, loop coverage and branch coverage are some of the coverage criteria for this testing. Though these criteria do not suggest the method of generating test cases according to some specific criterion, but they are helpful for evaluating a set of test cases. Using the white-box testing, one is able to look inside the box. The tester uses this internal understanding of the software at the time of choosing the test data.

Other names such as **Glass-box testing**, **Structural testing**, **Clear-box testing**, **Open-box testing**, **Logic-driven testing** and **Path-oriented testing** are also given to white-box testing.

The illustration of white-box testing has been given in **figure 4.6**. To select the test cases, the code is to be examined and not the specifications. In the white-box software testing approach, the test data are derived from the logic of the program after examining program structure. This testing is also sometimes known as **clear-box testing**.

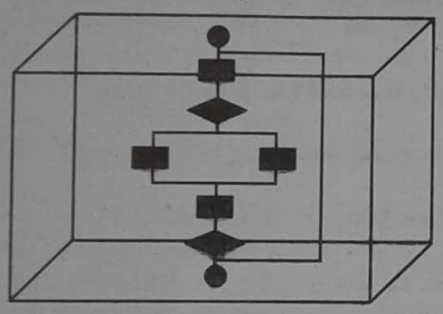


Figure 4.6: White-box Testing

Unlike black-box testing that is solely dependent on the understanding of the system requirements; white-box testing requires a detail internal knowledge of the program. It is evident from the software engineering literature that glass-box testing is the primary intent behind developing testing methodology because the prime concern is on internal structure of the programs.

However, numerous testing techniques that follow the black-box testing methodology have been developed because it has gained general consensus. One thing must be kept in mind that the testing techniques are implemented during the design phase of test and once they have been applied their influence is very hard to determine.

With the help of structural testing, it is not possible to find out whether any code has been omitted or not. However, it is possible to evaluate the adequacy of the test suite with regard to factors such as fault- finding ability or execution of the test suite components.

2) **Black-Box Testing/Functional Testing:** The structure of a program is not used in functional testing. Test cases are selected based on the specifications or the requirements of the program. The internal details of the program are not required for deciding the test cases. Functional testing is also referred to as 'black-box testing' because of its character.

Client participation is important in this testing as functional testing involves testing of all interfaces. The specifications of this test must be detailed as all features of the software system are tested. Questions pertaining to who, where, when, what and how with respect to conducting tests should be described in detail.

When clients are involved in a part of testing it is known as alpha testing. Alpha testing is closely monitored by the developers. They note how the clients use the system and what further improvements are required. Black-box testing is conducted to check the functional requirements of the system. This test helps the software engineer to originate a group of input conditions that will fulfil the entire functional requirements of the system.

There are certain other names used to refer Black-Box Testing (BBT) like **specifications testing, data-driven testing, behavioural testing, functional testing and input/output-driven testing.**

Black-box testing considers the system as a black-box because the tester does not have any idea about the internal functions of the item that is to be tested.

Black-box testing normally covers the following types of issues:

i) Erroneous functions,
ii) Data structure errors,
iii) Omitted functions,
iv) Performance errors,
v) Beginning and ending errors, and
vi) External database access errors.

The working of every module is tested with respect to the requirements and its events. Right input/output relationship is only studied. **Figure 4.7** depicts black-box testing.

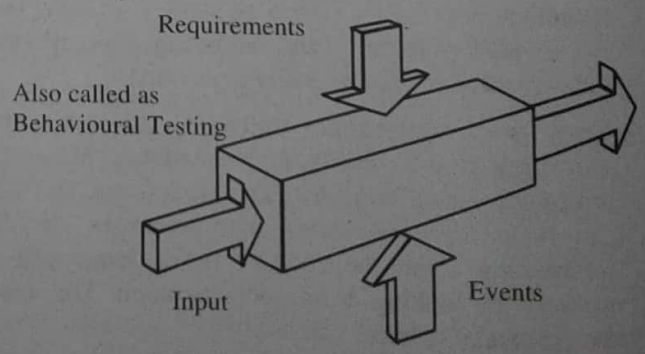


Figure 4.7: Black-Box Testing

For example, in black-box testing, the tester knows only what **inputs** are being given to arrive at certain outputs. He/**she** has no knowledge how the inputs are arriving at the given output. Therefore he/she does not detect programming code and does not require any further information about program except its specifications.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Ques 15) What are the advantages and disadvantages of white-box testing?

Ans: Advantages of White-Box Testing

1) The tester has to provide careful reasoning regarding testing.
2) The division done by execution equivalence is approximated.
3) Errors invisible in the code are exposed.
4) It is possible to optimise the code by discovering the hidden errors and eliminating such defects.
5) As the developers make a careful description of anything that has been newly implemented, the programmers are able to introspect it more closely.
6) The tests can be traced from the source. This allows to capture those changes made in the software in future with the help of changes in the test.
7) Automation of white box testing is easy.
8) It is possible to define clear, engineering-based rules that say when test must be stopped using white-box testing.

Disadvantages of White-Box Testing

1) Whether user requirements have been met accurately or not, is not ensured by white-box testing. As the code is not executed, it is impossible to say whether it will run correctly or not.
2) Using this test, it is not possible to determine whether the statements, paths, conditions and decisions while doing analysis are enough for the proposed requirements.
3) There are some situations where usage of checklists is dominated over white-box testing. Sometimes defects might get introduced directly into the work product if there are defects in the checklists. All the errors must be fully analysed.
4) The programmer must have a high level of knowledge as white-box testing is significantly complex in nature.
5) There are situations when it is not possible to test every condition that exists. Some conditions of the program will be left untested.
6) Some missing functionality might not be uncovered as the test takes into consideration the software as it exists.

Ques 16) Explain the following testing:

1) **Equivalence Partitioning**
2) **Boundary Value Analysis**
3) **Orthogonal Array Testing**
4) **Graph Based Testing**

Or

What are the different black box testing techniques?

Or

Discuss the guidelines for performing equivalence partitioning and boundary value analysis.

Ans: Black-Box Testing Techniques

This type of test can be applied in a number of ways:

1) **Equivalence Partitioning:** This is a black-box testing method which divides the program inputs into various classes of data. An equivalent class (represents a group of valid and invalid input values) is defined for specific input on the basis of which test case design is developed. Thus, equivalence partitioning attempts to define test cases which expose classes of errors which in turn decrease the number of test cases that should have been developed.

The domain of inputs is divided into equivalence classes. If a test in such an equivalence class is successful, then all tests in that class will be successful. In other words, we have to recognise classes of test cases in such a manner that success of a test case in a class indicates the success of other test cases. If such classes can be recognised then each class forms an equivalence class. As such, successful test in an equivalence class is equal to successful completion of complete testing of a program. The inputs are divided into equivalence classes based on its behaviour pattern in the module. In the similar groups, the pattern of module is considered to be different and for these, new classes form equivalence classes as shown in **figure 4.8.**

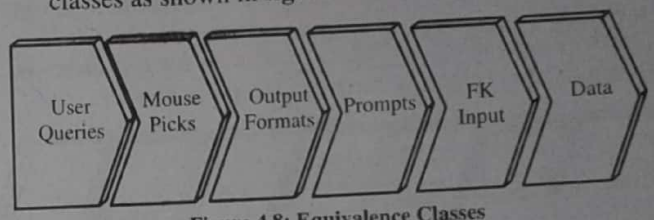


Figure 4.8: Equivalence Classes

It is assumed that if the specifications require the same behaviour from each element in a class of values, then the program will be class of values. This is the core idea of forming equivalence classes. **For example,** the determination of absolute value of integers from the specifications of a module states one behaviour for positive and another for negative integers. In this, two equivalence classes get created, one comprising positive integers and the other comprises negative integers.

Equivalence classes are also assigned to output. This can be achieved by creating test cases such that the output belongs in the output equivalence class. Output classes reveal more errors that may not be revealed by only input classes. It is more complex task to determine the test cases for output classes.

For example, database must be able to handle records from 1 through 16,383 as stated by specifications for a database.

If the database product can handle 34 and 14870 records then it may work well for 8534 records. If the product works well for one test case in 1 to 16,383, then it will work correctly for others within this range. This range comprises an equivalence class. There are three equivalence classes for this database product as shown below:

i) **Equivalence Class 1:** Less than one record.
ii) **Equivalence Class 2:** From 1 to 16,383 records.
iii) **Equivalence Class 3:** Greater than 16,383 records.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

One test class from each equivalence class is selected for testing the database product.

Guidelines for Equivalence Partitioning

The following rules define equivalence classes:

i) **Range:** When an input condition mentions a range, then one valid and two invalid equivalence classes are used.

ii) **Specific Value:** When an input condition needs a particular value then one valid and two invalid equivalence classes are used.

iii) **Member of a Set:** When an input condition is particular to a member of a group, then one valid and one invalid equivalence classes are used.

iv) **Boolean:** When an input condition is Boolean, then one valid and one invalid equivalence classes are used.

2) **Boundary Value Analysis:** Since, many errors occur at the boundaries of the input domain as compared to the center, hence Boundary Value Analysis (BVA) uses bounding values for test case selection. BVA is opposite to equivalence partitioning. In this, values are chosen at the extremes of classes for test cases as opposed to selecting the random input values.

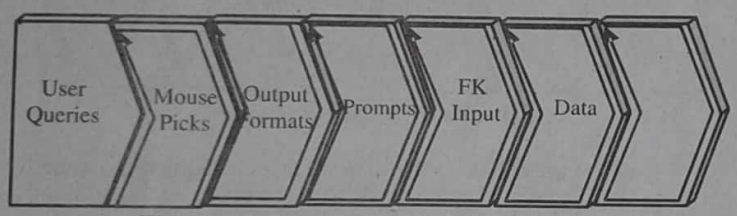


Figure 4.9: Boundary Value Analysis

In BVA, test case designers design test cases for extreme conditions in output. An effective test case is one which is successfully detecting faults which were not discovered earlier. Boundary value analysis is one such technique which maximises the chances to detect a new error or fault.

When test cases are selected on or one end of boundary of equivalence class, then the chances of finding the errors will be high. **For example,** while testing database product, following cases must be selected:

Test case 1	0 records	Member of equivalence class 1 and neighbouring to boundary value
Test case 2	1 record	Boundary value
Test case 3	2 records	Neighbouring to boundary value
Test case 4	723 records	Member of equivalence class 2
Test case 5	16,382 records	Neighbouring to boundary value
Test case 6	16,383 records	Boundary value
Test case 7	16,384 records	Member of equivalence class 3 and neighbouring to boundary value

This example applies to both input as well as output specifications.

Equivalence classes along with boundary value analysis are considered to be important technique that produces a small set of test data with the highest possibility of finding most errors.

Guidelines for Performing Boundary Value Analysis

i) **Range:** If an input is bordered by values x and y, then test cases must be created with the same values and values just above and below x and y.

ii) **Maximum and Minimum Numbers:** If an input condition has a number of values, test cases must be designed with the minimum and maximum numbers and values above and below the minimum and maximum.

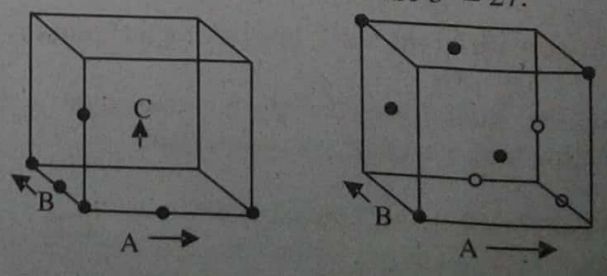
iii) **Output Conditions:** The above guidelines are to be applied to output conditions as well.

iv) **Internal Data Structures:** In internal data structures, test cases are designed to exercise it at its boundary.

3) **Orthogonal Array Testing:** Orthogonal Array Testing (OAT) is a two-dimensional feature in which selecting any two columns throws up every pair of each number present in the array.

It is category of a black-box testing technique which is used when the input domain of a program is small but at the same time too large for thorough testing. It is used in system testing, user interface testing, regression testing, configuration testing and performance testing. In OAT, a test case is represented by a row and a variable is represented by a column. Pair-wise factors are combined in OAT. It helps to design test cases which give maximum test coverage with few test cases.

Several advantages associated with OAT are like; it minimises the cycle time of testing, gives equally distributed coverage of test domain and produces lesser test cases because of summarising test set. However, there is one disadvantage that it does not provide assurance of widespread coverage of the test domain. To understand the difference between OAT and 'one input at a time' conventional approach, a system with three input items A, B and C are considered. Each item has three distinct values. The numbers of possible test cases are $3^3 = 27$.



One input items at a time (Orthogonal array)

Figure 4.10: Geometric View of Test Cases

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Figure 4.10 shows a geometric view of possible test cases related to A, B and C.

4) **Graph-Based Testing:** This is a testing technique where a graph is created first and then relation between causes and effects is established. Thus, it is also referred to as the **Cause Effect Graph.**

The input condition is the "cause" and the "effect" is the series of computations that are to be implemented. In cause effect graph, a node is formed by each condition.

There are **four basic symbols** that convey the inter-dependent relationship between the cause and effect as shown in **figure 4.11**.

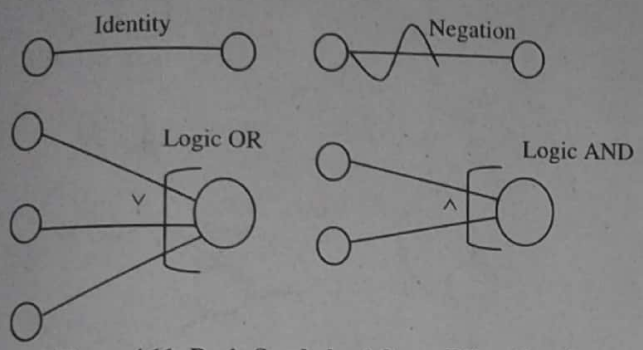
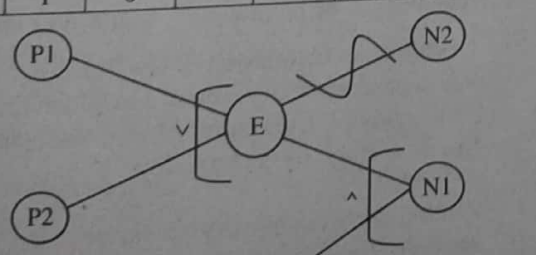


Figure 4.11: Basic Symbols of Cause Effect Graph

Let's take an **example** which shows the cause effect graph and the decision table. The first column character must be 'P' or 'Q'. The second column must be a number. If the first character is incorrect, then message A_{12} must be printed. If the second column does not have a number, then the message A_{13} should be printed.

Table 4.1: Cause Effect Graph Decision

1 = True, 0 = False						
Test Data	Causes			Effect		
	P1	P2	P3	N1	N2	N3
1	0	0	0	0	1	1
2	0	0	1	0	1	1
3	0	1	0	0	0	1
4	0	1	1	1	0	0
5	1	0	0	0	0	1
6	1	0	1	1	0	0



N1: Update the file
N2: Message A_{12}
N3: Message A_{13}

P1: First character is "P"
P2: First character is "Q"
P3: Second is a number

Figure 4.12: Cause Effect Graph for the Given Example

There are three input and eight possibilities. P1 = true and P2 = true is an impossible condition.

Before creating high level of test cases, cause and effect graphing helps in understanding the functionality of the system as the tester must be able to recognise the different causes and effect

Ques 17) What are the advantages and disadvantages of black-box testing?

Ans: Advantages of Black-Box Testing

1) Test conducted is impartial as the designer and tester are not dependent on each other.
2) Knowledge of programming languages is not necessary for tester.
3) Testing is accomplished according to the user's viewpoint, not according to designer.
4) Test cases can be designed just after the specification is finished.
5) The verification of inconsistencies in actual systems and specifications can be done using black-box testing.

Disadvantages of Black-Box Testing

1) This testing is based on requirements rather than design hence certain logical errors in coding as well as structure problems may be missed.
2) It may lead to redundant testing as requirements may execute the similar code repeatedly leading to the same test being conducted many times.
3) Large sample space is required for test inputs.
4) It is not always possible to recognise all inputs within the restricted testing time. Hence, writing test cases become slow and complex.
5) There is high likelihood of having unidentified paths while testing.

Ques 18) What is test case?

Ans: Test Case

A test case is a software testing document, which consists of event, action, input, output, expected result and actual result. Another definition of (IEEE 829-1998) a test case is an input an expected result.

The optional fields are a test case ID, test step or order of execution number, related requirement(s), depth/ test category, author, and check boxes for whether the test is manual or and has been automated.

A test case is a set of instruction designed to discover a particular type of error or defect in the software system by including a failure.

Test case design is a part of system and component testing where you design the test cases (input and predicted outputs) that test the system.

DOWNLOADED FROM KTUASSIST.IN
Scanned by CamScanner

Goal of the test case design process is to create a set of test cases that are effective in discovering program defects and showing that the system meets its requirements. One test case can give rise to many other tests.

Goal of selected test cases is to ensure that there is no error in the program and if there is it then should be immediate depicted. Ideal test casement should contain all inputs to the program. This is often called **exhaustive testing**.

There are two criteria for evaluating a set of test cases.

1) Generating a set of test cases that satisfy a given criterion.
2) Each test case needs proper documentation, preferably in a fixed format.

Example of Test Case Format

Test case name	Test Case ID
Purpose of test	Testing object (unit, application or module, etc.)
Test attribute	
Test focus (function, feature, process, interface, validation, verification, etc.)	
Test type (alpha, beta, unit, integration, system)	
Test process	Set of instructions for conducting the tests – Initial state condition – inputs –specifications – output expected.
Test results	Expected and actual and comparison, error description, post-process state.
Action	Correction, authorization and feedback through retest.
Action to initialize the pre-test status.	

Ques 19) Explain the basic path testing in detail.

Or

Write short note on the following:

1) **Flow Graph Notation**
2) **Cyclomatic Complexity**
3) **Graph Matrix**

Ans: Basic Path Testing

Basis path testing is a white-box technique. It allows the design and definition of a basis set of execution paths. The test cases created from the basis set allow the program to be executed in such a way as to examine each possible path through the program by executing each statement at least once.

Basic path testing enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

Following are to be necessary to understand the basic path testing:

1) **Flow Graph Notation:** To be able to determine the different program paths, the engineer needs a representation of the logical flow of control. The control structure can be illustrated by a flow graph. A flow graph can be used to represent any procedural design.

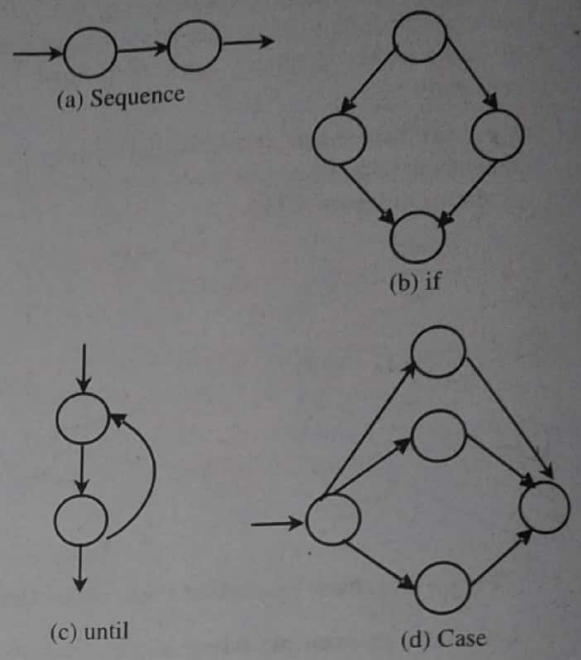


Figure 4.13: Flow Graph Notation

On a flow graph:

i) Arrows called **edges** represent flow of control
ii) Circles called **nodes** represent one or more actions.
iii) Areas bounded by edges and nodes called **regions**.
iv) A predicate node is a node containing a condition

Any procedural design can be translated into a flow graph. Compound Boolean expressions at tests generate at least two predicate node and additional arcs.

The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

2) **Cyclomatic Complexity:** Cyclomatic complexity is one such complexity measure in which the complexity of a module is the number of independent cycles in the flow graph of the module.

Cyclomatic complexity measures the amount of decision logic in a single software module. It is used for two related purposes in the structured testing methodology:

i) It gives the number of recommended tests for software.
ii) It is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Cyclomatic complexity is based entirely on the structure of software's control flow graph.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

Complexity is computed in one of three ways:

i) The **number of regions of the flow graph** corresponds to the cyclomatic complexity.

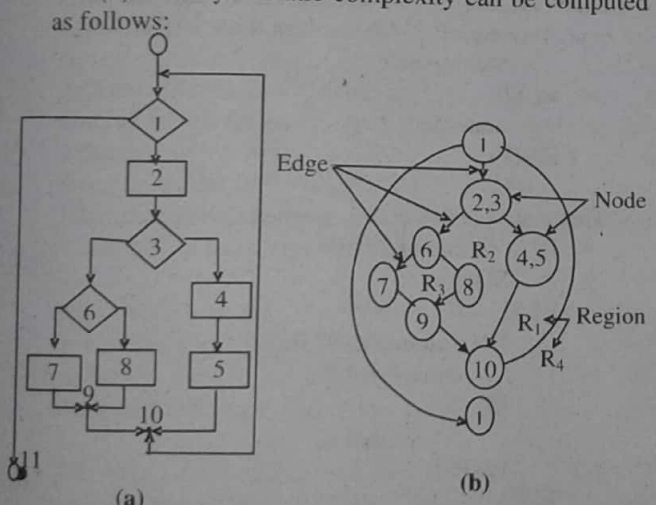
ii) Cyclomatic complexity, V(G), for a flow graph, G, is defined as:
$V(G) = E - N + 2$

Where E is the number of flow graph edges, N is the number of flow graph nodes.

iii) Cyclomatic complexity, V(G), for a flow graph, G, is also defined as
$V(G) = P + 1$

Where P is the number of predicate nodes contained in the flow graph G.

Let consider the flow graph as shown in **figure 4.14(b)**. The cyclomatic complexity can be computed as follows:



i) The flow graph has four regions
ii) V(G) = 11 edges – 9 nodes + 2 = 4
iii) V(G) = 3 predicate nodes + 1 = 4

Figure 4.14: Flowchart (A) and Flow Graph (B)

3) **Deriving Test Cases:** The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to derive the basis set:

i) Using the design or code as a foundation, draw a corresponding flow graph.

ii) Determine the cyclomatic complexity of the resultant flow graph.

iii) Determine a basis set of linearly independent paths.

iv) Prepare test cases that will force execution of each path in the basis set.

4) **Graph Matrix:** The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a **graph matrix**, can be quite useful.

A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.

A simple **example** of a flow graph and its corresponding graph matrix is shown in **figure 4.15**. Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes.

For example, node 3 is connected to node 4 by edge b.

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

i) The probability that a link (edge) will be executed.

ii) The processing time expended during traversal of a link.

iii) The memory required during traversal of a link.

iv) The resources required during traversal of a link.

To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in **figure 4.15** is redrawn as shown in **figure 4.16**. Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity). Represented in this form, the graph matrix is called a **connection matrix.**

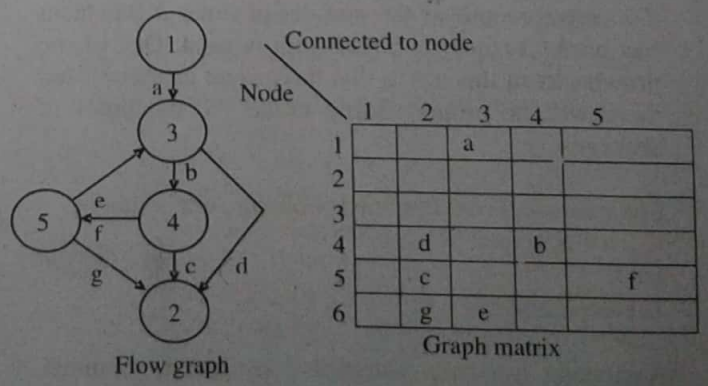


Figure 4.15.: Graph Matrix

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Node \ Connected to node	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1
						3 + 1 = 4

Graph matrix

Cyclomatic Complexity

Figure 4.16: Connection Matrix

In **figure 4.16**, each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity.

Ques 20) What do you understand by Control Structure Testing?

Or

Explain the following types of testing:

1) **Statement Coverage Testing**
2) **Branch Testing**
3) **Condition Testing**
4) **Data Flow Testing**
5) **Loop Testing**

Ans: Control Structure Testing

Control structure testing is a group of white-box testing methods. Control structure testing is used to enhance the coverage area by testing various control structures (which include logical structures and loops) present in the program. Basic path testing is used as one of the techniques for control structure testing.

Various types of testing performed under control structure testing are:

1) **Statement Coverage Testing:** This is considered as the simplest type of white-box testing. The test cases are run in such a manner that every statement is executed at least one time. In spite of the fact that this testing is achieved, it is not sufficient to show confidence in the behaviour of the product.

   To keep a record of the number of times a statement has been executed, a CASE tool is used. One of the drawbacks of this test is that there is no guarantee that there will be proper testing of all the outcomes of branches.

   For example, consider the following code segment:

   If(s > 1 && t == 0)
   x = 9;
   Test case: s = 2; t = 0;

   A mistake has been committed by the programmer here. The compound conditional statement (s >1 && t = = 0) must have been written as(s > 1 || t == 0).

   However, if the chosen value for x is 9, then the statement will be executed without demonstrating any errors.

2) **Branch Testing:** In this testing, a series of tests are used to ensure that all the branches have been tested at least one time. This technique is also known as **decision coverage**. The branch or statement coverage technique falls under the category of structural tests.

   In order to ensure that all the possible outcomes occur at least one time, sufficient test cases must be needed for every branch or program decision to be executed. For those applications that are highly integrated, the decision coverage is not alone sufficient. It is just considered to be the minimum level at which the coverage must be done for maximum number of software products.

   This example has two branches to be executed:

   ```
   IF (a equals b) THEN
       Statement 1
   ELSE
       Statement 2
   END IF
   ```

   This example also has just two branches to be executed, despite the compound conditional:

   ```
   IF (a equals b AND c less than d ) THEN
       Statement 1
   ELSE
       Statement 2
   END IF
   ```

   This example has four branches to be executed:

   ```
   IF (a equals b) THEN
       Statement 1
   ELSE
       IF (c equals d) THEN
           Statement 2
       ELSE
           Statement 3
       END IF
   END IF
   ```

3) **Condition Testing:** In this testing sufficient number of test cases must be assigned to each condition of a program so that all the possible outputs can be taken at least one time. When more than one condition has to be estimated in order to reach a decision then it varies from branch coverage testing. A sufficient number of test cases must be needed for multi-condition coverage so that every combination of conditions can be exercised.

   It is a test method that exercises the logical conditions contained in a program module. A simple condition is a Boolean value, such as True or False. These simple conditions concatenate with relational operators. A relational repression takes the form:

   e1 < relational – operator > e2

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

where, e1 and e2 are arithmetic expressions and < relational – operator > is one of the following:

i) <
ii) <=
iii) =
iv) NOT =
v) > or > =

A compound condition is composed of two or more simple conditions with Boolean operators, and parentheses. We assume that Boolean operators allowed in compound condition include OR (/) AND (&) and NOT (–). A condition without relational expressions is referred to as a Boolean expression. If a compound condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

4) **Data Flow Testing:** This technique is also known as **data flow guided testing**. The points at which values are received by variables and the point at which the value is used is the prime focus of this testing methodology. There are many authors who consider it as the path testing because it can be used to check the reality of path testing. To ensure that each possible data flow has been executed at least one time, enough number of test cases is required in this technique. The sequence of variables and actions performed on the program paths are studied using data flow analysis. This technique can be used and implemented as both dynamic and static test methodologies.

Data flow testing is best suited for detecting misspelled identifiers and wrong uses of constants and variables. But it is impossible to identify missing statement using the data flow testing.

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameter or global variables. For a statement with S as its statement number;

DEF (S) = {X/Statement S contains a definition of X}

USE (S) = {X/Statement S contains a use of X}

If statement S is an if or loop statement, then its DEF set is empty and its USE set is based on condition of statement S. the definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 that contains no other definition of x

A Definition – Use (DU) chain of variable X is of the form [X, S, S1], where S and S1 statement numbers, X is in DEF (s) and USE (S1), and the definition of X in statement S is live at statement S1. One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program, because the corresponding branch has no definition of any variable.

5) **Loop Testing:** If this criterion is used, enough number of test cases must be needed so that each loop in the program can be run for zero, one, two and multiple times. This includes initialisation, terminating condition and normal running of the loop.

The validity of the loop constructs is the prime focus of loop testing.

Four different classes of loops can be defined:

i) **Simple Loops:** The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop:
   a) Skip the loop entirely,
   b) Only one pass through the loop,
   c) Two passes through the loop,
   d) m passes through the loop where m < n, and
   e) n – 1, n, n + 1 passes through the loop.

ii) **Nested Loops:** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. **Beizer** suggested an approach that will help to reduce the number of tests.
   a) Start at the innermost loop. Set all other loops to minimum values.
   b) Conduct simple loop test for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Add other tests for out-of-range or excluded values.
   c) Conduct tests for the next loop, but keep all other outer loops at minimum values and other nested loops to typical values.
   d) Continue until all loops have been tested.

iii) **Concatenated Loops:** These loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop1 is used as the initial value for loop2, then the loops are not independent. When the loops are not independent, then the approach applied to nested loop is recommended.

iv) **Unstructured Loops:** Whenever possible, this class of loops should be re-designed to reflect the use of structure programming constructs.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

In **figure 4.17**, four types of loops have been illustrated. They are simple loops, nested loops, concatenated loops and unstructured loops. For several types of loops, exclusive test case must be designed.

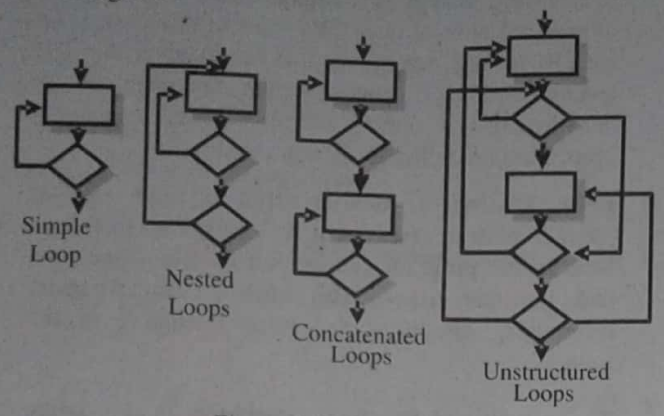


Figure 4.17: Loop Testing

Ques 21) What are the program analysis tools?

Or

What are static testing strategies? List out the different static strategies.

Or

What are dynamic testing strategies?

Ans: Program Analysis Tools

These tools usually use an automatic tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of a program like size, complexity, compliance with programming standards etc.

These tools can be classified into two categories

1) **Dynamic Analysis Tools / Dynamic Testing Strategies:** These techniques require the program to be executed and its actual behavior recorded. A dynamic analyser instruments the code. An instrumented code when executed allows recording the behavior of the software for different test cases.

   After the software has been tested with its full test suit and its behavior recorded then the dynamic testing tool carries out a post execution analysis and produces reports which described the structural coverage that has been achieved by the complete test suit.

   Normally the results of this testing are produce by means of histogram or pie chart to describe the structural coverage achieved for different modules of the program. It helps to eliminate redundant test cases.

2) **Static Analysis Tools / Static Testing Strategies:** Static testing is the systematic examination of a program structure for the purpose of showing that certain properties are true regardless of the execution path the program may take.

   Consequently, some static analyses can be used to demonstrate the absence of some faults from a program. Static testing represents actual behavior with a model based upon the program's semantic features and structure.

   Human comparison often consists of people exercising little discipline in comparing their code against notions of intent that are only loosely and imprecisely defined. But human comparisons may also be quite structured, rigorous, and effective as is the case of inspections and walkthroughs, which are carefully defined and administered processes orchestrating groups of people to compare code and designs to careful specifications of intent.

Types of Static Strategies

i) Formal Technical Reviews,
ii) Code Review
   a) Code Walk-through
   b) Code Inspection,
iii) Compliance with Design and Coding Standards. (Coding Standard and Guidelines)

Ques 22) What is a formal technical review?

Or

Explain the peer reviews? What are the objectives of formal technical review?

Or

What is meant by "Formal Technical Review"? Should it access both programming style as well as correctness of software? Give reasons.

Ans: Formal Technical Reviews (Peer Reviews)

Technical review should be done by the team of members. The document, which is going to be reviewed, who has prepared and reviewers should sit together and do the review of that document. It is also called **Peer Review.** If it is a technical document, it can be called as **formal Technical review**. It varies depends on the company policy.

Reviews are an effective method for quality control and tend to enhance productivity. Different members (players) which are included in the review group are the author of the product under review (producer), a quality control engineer, a moderator (key person responsible for ensuring that the review proceed smoothly), a recorder, and reviewer and/or user representative shown in **figure 4.18**.

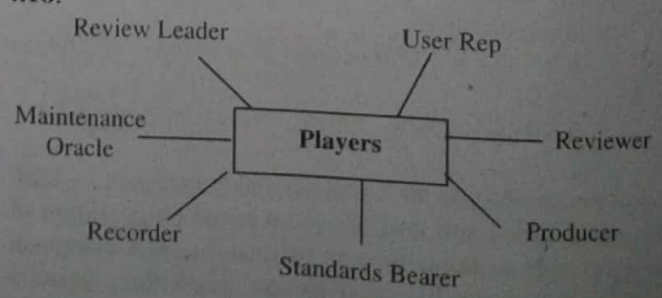


Figure 4.18

DOWNLOADED FROM KTUASSIST.IN
Scanned by CamScanner

FTR access both programming style as well as correctness of software because it is good but more information needed. During FTR, a reviewer actively records all issues that have been raised. These are summarized at the end of the review meeting and a review issues list is produced. In addition, a formal technical review summary report is completed. A review summary report answers three questions:

1) What is reviewed?
2) Who reviewed it?
3) What were the findings and conclusions?

Objectives of Formal Technical Review

1) To uncover errors in logic or implementation,
2) To ensure that the software has been represented accruing to pre-defined standards,
3) To ensure that software under review meets the requirements,
4) To make the project more manageable.

For the success of formal technical review, the following are expected:

1) The schedule of the meeting and its agenda reach the members well in advance,
2) Members review the material and its distribution,
3) The reviewer must review the material in advance.

Ques 23) What is code reviews? Discuss the types of code reviews.

Or

Explain the following:

1) **Code Walk-through**
2) **Code Inspection**

Ans: Code Reviews

Code reviews are extremely cost-effective strategies for reduction in coding errors in order to produce high quality code. Review process was started with the purpose of detecting defects in the code. Though design reviews substantially reduce defects in code, reviews are still very useful and can considerably enhance reliability and reduce effort during testing.

Code reviews are designed to detect defects that originate during the coding process, although they can also detect defects in detailed design.

Code reviews are usually held after code has been successfully completed and other forms of static tools have been applied but before any testing have been performed. Therefore, activities like code reading, symbolic execution, and static analysis should be performed, and defects found by these techniques corrected before code reviews are held. The main motivation for this is to save human time and effort, which would otherwise be spend detecting errors that a complier or static analyser can detect. In other words, the entry criterion for code review is that the code must compile successfully and has been "passed" by other static analysis tools.

Documentation to be distributed to the review team members includes the code to be reviewed and the design document. The review team for code reviews should include the programmer, the designer, and the tester. The review starts with the preparation for the review and ends with a list of action items.

Types of Code Reviews

1) **Code Walk-through:** Code walk-through is an informal code analysis technique. After elimination of syntax errors members of the development team are given the code for few days before the walk-through meeting to read and understand the code. Members note down their findings to discuss them in a walk-through meeting where coder is also present.

   The main objectives of code walkthrough (hand simulation of code execution) are to discover the algorithmic and logical errors in the code.

   Guidelines for Code Walk-Through

   Some guidelines for code walk-through are as under:

   i) Code walkthrough team should be optimal, i.e., it should not be too big or too small.
   ii) Meeting discussions should focus on discovery of errors rather than how to fix these errors.
   iii) Managers should not attend the walk-through meeting in order to foster cooperation and any bad feeling among team members.

2) **Code Inspection:** In contrast to code walkthrough, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.

   In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walkthroughs.

   For example, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for this kind of mistakes in the code, rather than by simply hand simulating the execution of the procedure.

   In addition to detecting the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics to identify the type of errors most frequently committed by their engineers. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

   The following is a list of some classical programming errors, which can be checked for during code inspection:

   i) Use of un-initialized variables,
   ii) Jumps into loops,

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

iii) Non-terminating loops,
iv) Incompatible assignments,
v) Array indices out of bounds,
vi) Improper storage allocation and de-allocation,
vii) Mismatches between actual and formal parameters in procedure calls,
viii) Use of incorrect logical operators or incorrect precedence among operators,
ix) Improper modification of loop variables,
x) Comparison of equality of floating point values, etc.

Ques 24) What are the different testing strategies? List them.

Ans: Testing Strategy
A test strategy provides the planning guidelines of the testing method of the software development life cycle. It is generated to assist project managers, testers and developers regarding principal issues which are testing objectives, methods of testing new functions, total time and resources needed for the project and environment for testing.

Test strategies help in answering certain questions such as how product risks are diminished at the test level, which testing must be done and which entry and exit criteria are applicable. They are created on the basis of design documents of development.

Software testing strategy acts as a guiding plan for quality assurance organisation, software developer, and customer. It lays down a set of milestones to measure manager progress. Problems should also surface at the earliest. It also offers guidance to practitioner. Software testing strategies consist of various levels which are to be performed for testing the software.

The first level tests single units of software. After this, these units are combined and can be used for checking interfaces. The final stage is testing the complete software to ensure that output is as per user requirements.

Figure 4.19 shows four levels of software testing:
1) Unit Testing,
2) Integration Testing,
3) System Testing, and
4) Validation/Acceptance Testing.

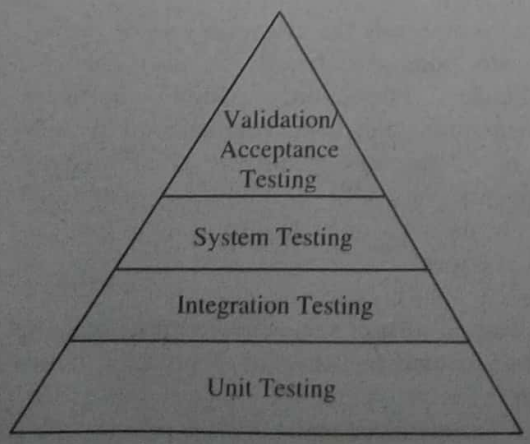


Figure 4.19: Levels of Software Testing

Ques 25) What is unit testing? Also discuss the types of unit testing.

Ans: Unit Testing
Once a module has been coded and studied successfully, unit testing comes into action. This testing is used for verification of the smallest unit (software component or module) of software design.

Necessary control paths are tested to discover errors within the module boundary with the help of the component-level design description. It is a white box testing technique which can be carried out in parallel for more than one component.

In this testing, the software engineer tests whether information flows into and out of the program module being tested. It tests individual programming unit by using the white box method. These units are called modules or atomic modules which are the smallest programming units.

Unit test is a group of path tests which are conducted to conclude that all paths of the program are fool-proof and will never end the program abruptly or will give any other adverse results.

Figure 4.20 shows the unit tests that are conducted. Module interface is tested. While testing the interface of module, the local data structure is also tested to guarantee that temporarily stored data maintains its integrity while implementation of all steps in an algorithm takes place.

Boundary conditions are also examined to ascertain smooth functioning of module at boundaries which are created to limit the processing.

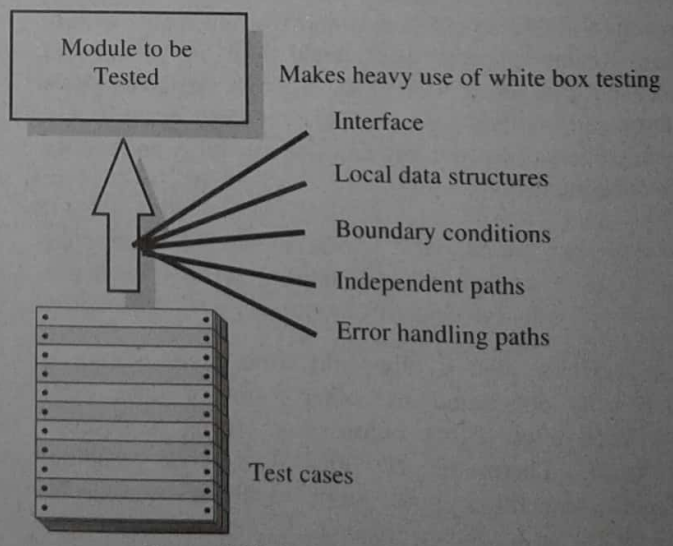


Figure 4.20: Unit Testing

Types of Unit Testing
Unit testing involves a series of individual tests where each test inspects an individual component that is new or modified. This is also referred to as **module test** as it tests separate units of code individually which makes up a

DOWNLOADED FROM KTUASSIST.IN
Scanned by CamScanner

program and ultimately the software. There are many tests that comprise unit testing:

1) **Module Interface:** This is examined to check data flow into and out of the 'unit' under review. The data flow across a module interface is tested first before performing other tests.
2) **Local Data Structure:** These are examined to check that temporarily stored data maintains its integrity during the implementation of an algorithm.
3) **Boundary Conditions:** This is examined to check that the module works well within restricted boundaries.
4) **All Independent Paths:** In this test, entire control structure is exercised and confirms that all statements in a module have been executed minimum once.
5) **Error Handling Paths:** Error-handling paths are tested after completion of other tests.

Ques 26) What are drivers and stubs?
Or
What are drivers and stub modules in the context of unit testing of a software product? Why are stubs and drivers module required?

Ans: Need of Drivers and Stub Module

There are two kinds of imitators, namely **drivers** and **stubs**. A driver is an active imitator that generates input messages for the real objects (units) under test. A stub is a passive imitator that accepts the output messages generated by the objects under test. Stubs can also send replays that are expected from the objects they are imitating. One can construct more complex imitators that act as both drivers and stubs.

Drivers and Stubs in Unit Testing

Unit tests may be designed before coding or once code has been created. After reviewing this design, test cases are created to check the errors in different units. The two modules, **drivers** and **stubs** are taken to test units independently:

1) **Driver:** A driver can be said to be a principal program which accepts test data, passes the data to the component being tested and provides the appropriate results.
2) **Stubs:** This replaces modules that are secondary components which are to be tested. It uses the secondary module's interface, performs data management, and provides verification and gives back control to the module being tested.

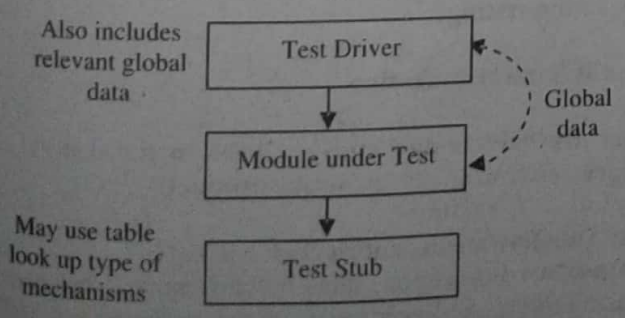


Figure 4.21: Test Drivers and Test Stubs

The unit-test environment is illustrated in **figure 4.22**.

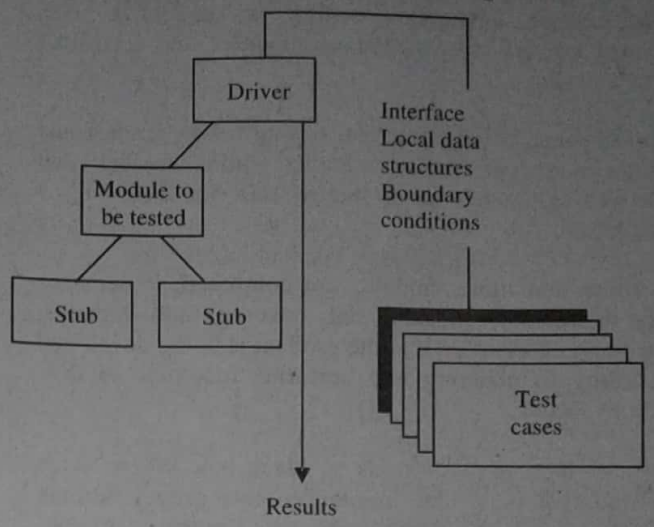


Figure 4.22: Unit-Test Environment

In most applications a driver is nothing more than a "main program" that accepts test-case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (called by) to the component to be tested.

A stub uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designated. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Ques 27) What is integration testing? List out the different integration testing.

Ans: Integration Testing

Integration testing is an organised technique and can be used in creating a program structure while simultaneously performing tests to reveal errors related to interfacing. It considers unit tested components and constructs a program structure guided by the design.

Program is designed and tested in small batches, where fault can be separated and corrected easily. The interfaces are tested wholly and an organised test approach can be used.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

In order to ensure that the product meets the specifications, integration testing is conducted. This ensures correct combination of modules and interfaces between them.

The movement from structural testing towards functional testing starts gradually with integration testing that considers a module as a hidden tool for executing a function.

As more and more modules are combined, it becomes very difficult to consider path coverage and domains. One must be satisfied that the product is being developed according to planning and performs functions as they were expected.

The product is treated as a black-box where it is verified that particular inputs generate proper outputs without taking into consideration the internal structure of the software. In functional tests, one should select test data which shows necessary or uncommon situations.

Types of Integration Testing

This test is being used for testing multiple modules which function together. The basic types of integration testing are as follows:

1) Big-Bang Integration Testing
2) Top-Down Integration Testing
3) Bottom-Up Integration Testing
4) Mixed Integration Testing
5) Regression Testing
6) Smoke Testing

Ques 28) Explain the following integration testing:

1) **Big-Bang Integration Testing**
2) **Regression Testing**
3) **Smoke Testing**

Ans: Big-Bang Integration Testing

In this technique, all modules are taken together and tested, i.e., all modules of a system are combined in an individual step. As such, this can only be applied for very small systems. One of the major problems of this technique is that detecting the location of error is difficult as it may belong to any modules which are to be combined.

The big-bang approach is a simple approach where all modules are examined separately and once completed they are all integrated simultaneously.

Regression Testing

In integration testing, a new module is added at every level. This may cause changes to the software as new data flow paths get created, new I/O may occur or new control logic is raised. These changes may create problems in functions that worked perfectly before. With regard to integration testing, regression testing refers to re-execution of modules that have already been tested earlier to make sure that changes have not brought about any unwanted side effects.

Regression testing is the process in which subset of earlier executed integration and test cases are examined again to guarantee that program changes have not brought degradation in the system.

This test considers effects of new changes on formerly integrated code. Complexity occurs when errors made in integrating new functions change the formerly examined functions, which is normal in big systems.

This test can be done either manually or by automated tools. It includes putting selected test cases into a regression bucket which is executed at regular intervals to detect problems caused by regression.

Smoke Testing

'Smoke test' refers to capability of a device to ensure that it does not initiate smoking, which represents a major problem.

Smoke testing is generally used whenever 'shrink-wrapped' software products are being generated. It helps in speeding up time-critical projects which aids the software team to review the project at regular intervals.

The most prominent feature of a smoke test is that it executes rapidly within a few minutes, hence providing a faster feedback as compared to executing full suites which may be a longer process (take hours or days).

Smoke test carried out on a specific build is also referred to as build verification test.

This test can be carried out either manually or automatically. If automated tools are used, then tests are begun with the similar process that produces the build as well.

For example, smoke test examines elementary questions such as "Does the program execute?", "Does it open a window?" or "Does pushing the major button do something?" The objective of this test is to ascertain whether the application is so poorly divided which makes next testing unnecessary.

Ques 29) What is top-down, bottom-up and mixed integration testing?

Or

What is Sandwich Testing?

Or

What are drivers and stub modules in the context of integration testing of a software product?

Ans: Top-Down Integration Testing

In top-down integration, modules are created and tested from the top to the lower levels of programming hierarchy (**figure 4.23**).

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

As each level is preceded by the previous level, it is considered to be an incremental approach. This can be conducted in 'depth' or 'breadth' manner:

1) **Depth:** This means that it is headed from the top level to the lowest level.

2) **Breadth:** This means that it starts from the top level and then moves to the next level. All modules at this level are examined before moving onto the next level.

Top down integration helps to create a complete skeleton of the system.

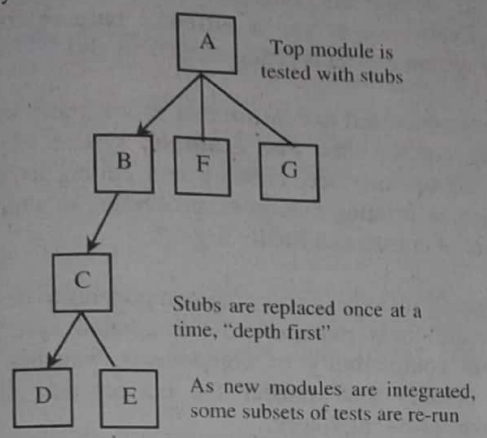


Figure 4.23: Top-Down Integration

Bottom-Up Integration Testing/Drivers and Stubs in Integration Testing

In bottom-up integration, the bottom level modules are created and examined at first individually and then in groups by test drivers. Bottom-up approach guarantees that every module is completely examined before they are used by calling module. This approach is opposite to the top-down approach and helps in early detection of error in critical modules.

In this technique, the lower level modules are first constructed and then tested continuing to upper level of the hierarchy. As modules at the lower levels are very specific, several modules are integrated to form a cluster or a build. These builds are then tested by test drivers which are written and put in place. **Figure 4.24** helps to understand the bottom-up approach.

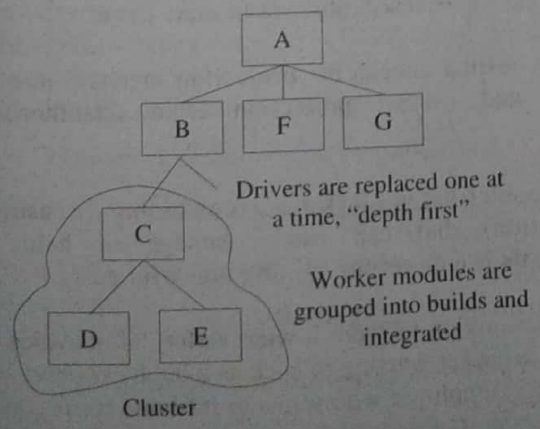


Figure 4.24: Bottom-up Integration

A bottom-up integration strategy may be implemented with the following steps:

1) Low-level components are combined into clusters (sometimes called builds) that perform specific software sub-functions.
2) A driver (a control program for testing) is written to coordinate test case input and output.
3) The cluster is tested.
4) Drivers are removed and clusters are combined moving upward in the program structure.

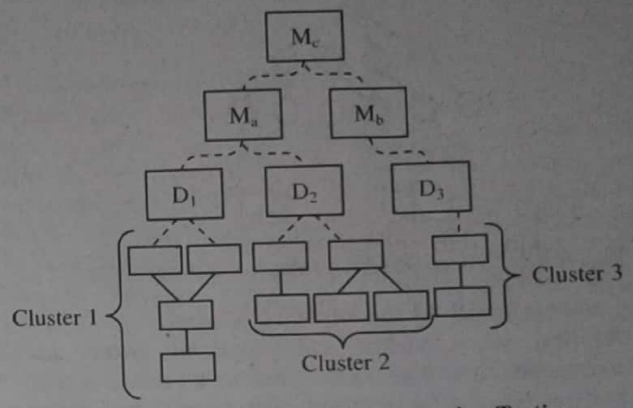


Figure 4.25: Bottom-Up Integration Testing

Integration follows the pattern illustrated in **figure 4.25**. Components are combined to form clusters 1, 2, 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c and so forth.

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top-down, the number of drivers can be reduced and integration of clusters is greatly simplified.

Sandwich/Mixed Integration Testing

Sandwich testing is also referred to as **Mixed Integration** testing because it combines both the top-down and bottom-up approaches.

Top-down approach can be initiated once the top-level modules have been coded and tested. Similarly, bottom-up approach can be initiated once bottom-level modules are ready for use. However, in mixed approach, testing can begin when modules are ready. This approach overcomes the problems caused by top-down and bottom-up approach.

In the top-down approach, one module gets added at a time until the entire tree like structure is formed. Thus, there is no need for drivers. Similarly, the bottom-up approach has no requirement of stubs. A sandwich strategy executes from top and bottom simultaneously and

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

meets at the mid-point. **Figure 4.26** depicts the three approaches:

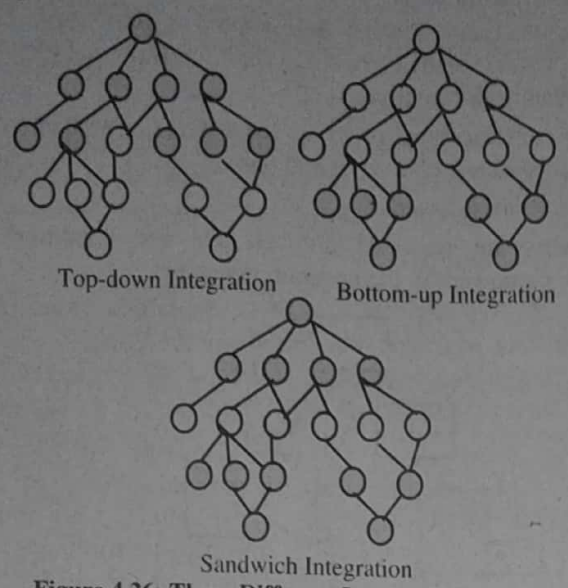


Figure 4.26: Three Different Integration Approaches

Integration is guided by the following principles:

1) At first, when module and integration testing are overlapped, it will be noticed that not all modules will be ready for integration simultaneously. First, subsystems of great concern should be put together for integration. If user interface and control were the most difficult part of the software, then the previous prioritisation will choose the top-down approach. This will happen if the customer wants to see a running program at the testing phase.

2) If machine interface and performance are important, then bottom-up integration approach will be used. A higher degree of concurrency will be maintained with bottom-up integration.

The top-down integration focuses on whether things will work successfully or not whereas bottom-up integration proves that things actually do work.

Ques 30) What is difference between top-down and bottom-up testing?

Ans: Difference between Top-Down Testing and Bottom-UP Testing

Table below shows the difference between top-down and bottom-up testing:

Table 4.2: Top-Down vs. Bottom-Up Testing

Top- Down Testing	Bottom-Up Testing
Top-Down testing conducted from main-module to sub module.	Bottom-Up testing conducted from sub module to main module.
If sub module is not developed a temporary program called Stub is used for simulate the sub module.	If main module is not developed a temporary program called Driver is used to simulate the main module.
Top -Down testing good if major flaws occur toward the top of the program.	Bottom-Up Testing good if major flaws occur toward the bottom of the program.
In this test condition difficult to create.	In this test condition easy to create.
Observation of test output is more difficult.	Observation of test output is easier.

Ques 31) What is system testing? List out the different types of system testing.

Ans: System Testing

System testing refers to a sequence of tests whose main aim is to completely implement the computer-based system. Every test serves a different purpose for the different system elements (integrated correctly).

This test is conducted to confirm that an integrated system fulfils its requirements. **For example,** system test will check a log-on interface, creating and editing an entry, transferring or printing outcomes, processing of summary or removal of entries and finally log-off.

System testing includes integrating components to develop a system and next examines the integrated system. It checks the compatibility of components, whether they interact correctly and transfer the correct data at the correct time across interfaces.

Types of System Testing

The different types of system testing are shown below:

1) Security Testing
2) Reliability Testing
3) Performance Testing
4) Recovery Testing
5) Thread Testing

Ques 32) What is security testing in system testing?

Ans: Security Testing

Security testing refers to testing that is carried out to maintain security of the system. It is done to ensure that unauthorised persons or other systems do not get access to the system and information.

Programs which allow access with password as well as other organisational security measures are tested.

Security testing checks the protection mechanism of a system and system protection from unauthorised access.

A computer-based system (containing sensitive information) that can cause damage or help to individuals is a target for gaining unlawful entry.

Illegal penetration includes a wide variety of activities. It could be a hacker wanting to hack as a hobby or sport and an unhappy employee who wants to infiltrate for revenge. Some dishonest persons also wish to access the system for illegal personal benefits.

DOWNLOADED FROM KTUASSIST.IN
Scanned by CamScanner

According to Beizer, "The system's security must, of course, be tested for invulnerability from frontal attack – but must also be tested for invulnerability from flank or rear attack."

In security testing, the tester acts as the hacker who wishes to penetrate the system.

Ques 33) What is reliability testing?

Or

What are the different parameters involved in reliability testing?

Ans: Reliability Testing

The probability of a failure-free system operation is referred to as software reliability. Apart from the many features of software, it also includes the testing process. It is difficult to directly evaluate software reliability by measuring the related factors.

Black-box testing is a sampling method for determining software reliability and also helps in locating failure data. An estimation model can be taken for further analysis of data to determine the present and future reliability.

Dependable software means the software that does not fail in sudden or catastrophic ways.

Software testing technique exposes the software's capability to function under environmental situations steadily which helps to discover problems pertaining to software design and working.

According to ANSI, "Software Reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is not a direct function of time.

Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its lifecycle.

Software will not change over time unless intentionally changed or upgraded."

Parameters Involved in Reliability Testing

1) Reliability testing consists of following dependent elements:
   i) Likelihood of failure-free operation,
   ii) Time range of failure-free operation, and
   iii) Environment in which it is carried out.
2) Reliability testing measures the following key parameters:
   i) **MTTF**(Mean Time To Failure)
   ii) **MTTR**(Mean Time To Repair)
   iii) **MTBF**(Mean Time Between Failures)

   MTBF = MTTF + MTTR

Ques 34) What is performance testing?

Or

Discuss the testing for performance?

Or

Write short notes on the following testing:

1) **Load Testing**
2) **Stress Testing**
3) **Soak Testing**
4) **Spike Testing**
5) **Configuration Testing**
6) **Isolation Testing**

Ans: Performance Testing

Performance testing is conducted to measure how quickly one feature of a system performs under a specific workload. Other features of the system like scalability, reliability and use of resources can be verified by this testing. This is a subset of performance engineering which tries to introduce performance in the design and architecture of a system before starting actual coding.

Performance testing can be used for different purposes such as how system fulfils performance criteria. It may compare the performance of two systems and decide which one is better. It can also detect the bad parts of the system and functions which reduces the performance of the system.

Performance objectives are decided at the planning stage and performance testing is used to ensure that these objectives are being met. These tests are run at the starting stage of development to get a comparison between actual and estimated statistics.

The different tests which help in measuring functionality and performance of a system are as follows:

1) **Load Testing:** Load testing is the simple performance test. It is used to understand the performance of a system with respect to a specific load.
2) **Stress Testing:** Stress test is conducted to understand how much load the system will be able to handle which will be above the expected maximum, i.e., whether the system will perform sufficiently or not, if present load goes up. This understands the upper limits of capacity of the system and verifies the system's strength in terms of extreme load.
3) **Soak Testing:** Soak testing is also referred to as **endurance testing** which checks whether the system can bear incremental expected load or not. Memory utilisation is also observed for determining possible leaks. Performance degradation is also an important concern which needs to be monitored to check whether throughput and/or response time after long period of continuous action is the same or better as compared to starting of test. The main aim of this test is to determine how system reacts after continuous use.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

4) **Spike Testing:** Spike testing is conducted by increasing the load very suddenly by several users at once and then monitoring the reaction of the system. The aim of this testing is to understand whether performance will be hampered, the system will stop working or will it be able to manage radical changes in load.

5) **Configuration Testing:** This testing monitors system reaction (performance and behaviour) to configuration changes to the system's components. **For example,** investigating with diverse load-balancing methods.

6) **Isolation Testing:** This refers to a test that repeats those tests which gave rise to a system problem earlier. These tests separate and guarantee fault domain.

Ques 35) What is recovery and thread testing?

Ans: Recovery Testing

This kind of testing is used to force the software to fail in different ways and checks that recovery is correctly achieved.

When the system performs automatic recovery, i.e., the system repairs itself, then re-initialisation, check-pointing mechanisms, data recovery and restart are assessed for accuracy.

If recovery is done manually, i.e., aided by human, then the Main-Time-to-Repair (MTTR) is assessed to find whether it is within satisfactory boundaries.

This test uses test cases that have been designed to assess how fast and easy the recovery of the system is done from a failure like blown circuit, power shut down, interface failure, disk crash, insufficient memory, etc.

A system should be such that it is capable of doing fast recovery with least human involvement. It must have a record of events before crashing and a record of messages when the system fails and restarts.

There are many computer-based systems which recover the system from errors and resume the functioning of system within a specified time. In some cases, systems should be fault tolerant which means that processing the faults must not stop the entire system function. In other cases, faults must be rectified within a time limit or major economic damages may take place.

Thread Testing

Thread testing technique is best suited for real-time systems. In this testing, the system or object processes through the processing of every external thread at every step. This test recognises and implements every probable processing 'thread'.

This testing tests the system reaction when executing threads through the system.

The thread based testing approach:

1) Combines a set of classes which respond to one input/event,
2) Combines and tests every thread separately, and
3) Detects the side effects.

Ques 36) What is validation testing?

Or

What is acceptance testing? What are the different phases of acceptance testing?

Or

What is alpha and beta testing of products?

Ans: Validation/Acceptance Testing

In order to check whether the software confirms the entire functional, behavioural and performance criteria, **validation** or **acceptance** testing is conducted.

Acceptance testing might engage chemical, physical or performance tests. It also checks whether requirements according to contract or specification are fulfilled or not.

According to International Software Testing Qualifications Board (ISTQB), "Acceptance testing is formal testing with respect to user needs, requirements and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorised entity to determine whether or not to accept the system".

In this type of testing, the software is reviewed by either a developer or a user. This test helps the users to analyse themselves whether the system is fulfilling their requirements or not.

In validation testing, a set of data is fed into the software as input. Before conducting this test, the probable outcomes must be known so that the test results can be compared with them.

If the output matches with the estimated result, then the user accepts the software. But, if there is a mismatch between them, then the user rejects the software.

Phases of Validation Testing

Validation testing is performed in two phases as illustrated below:

1) **Alpha Test:** Alpha test is carried out by the customer at the developer's site under the supervision of the developer. In this test, all versions of the software are tested.

   This test is conducted before the software is commercially made available to users. This test is usually carried out using white-box testing followed by black-box and grey-box testing.

   The goal here is to replicate real user tasks and operations by using such technique.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

The actual alpha testing takes place in an environment which looks like a lab. Once such techniques are successfully done, this testing has become completed.

2) **Beta Test:** Beta test is performed by the end user at the customer's site and does not involve the developer. When a fault is detected during test, it is reported to the developer who makes necessary changes and next releases the new version to the customer.

Beta testing comes after alpha testing and people external to the organisation are engaged in this test. The goal of this test is to find out errors before commercial release of the product so the software is distributed among very limited number of users outside the organisation.

Outsourced testing companies are hired for this testing as their feedback is unbiased and provides a different view as compared to employees of the software development company.

This feedback helps in fixing the defects that were earlier undetected. It also helps in preparation of support teams to face predictable issues or enforce modifications to functionality at the last moment.

Sometimes, the beta version of software is made available to public to get 'real-world' information for software that depends on performance acceptable to work in a proper manner.

Ques 37) What is difference between alpha and beta testing?

Ans: Differences between Alpha and Beta Testing

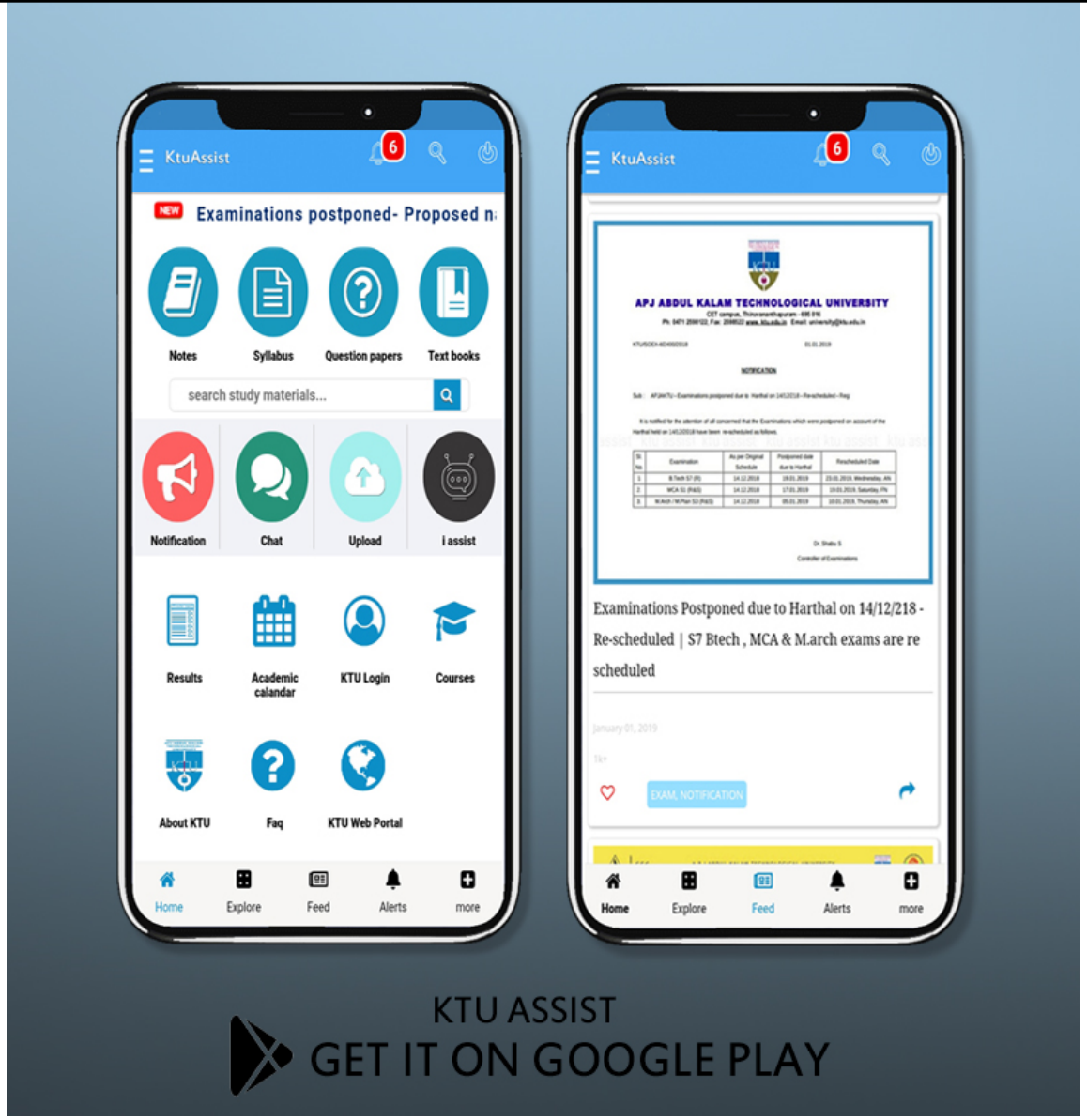
The difference between alpha and beta testing are stated in **table 4.3**:

Table 4.3: Differences between Alpha and Beta Testing

Alpha Testing	Beta Testing
1) Alpha testing is carried out after system testing where the manufacturer wants to show the application to customer before delivery.	Beta testing is carried out after alpha testing.
2) Alpha testing is performed by customer representative or by tester in the presence of customer representative.	Beta testing is performed by customers. Testers may be present to support them.
3) It is carried-out in development environment.	It is performed in production/semi-production environment.
4) Data used may be defined by testers with the help of customer or may be shared by customer.	Production data is generally used.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner



END

facebook.com/ktuassist

instagram.com/ktu_assist