I. Static White Box Testing

1. Examining the Design and Code

1. Static white-box testing is the process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it.

2. It's sometimes referred to as structural analysis.

3. Advantage of static white-box testing is that it gives the team's black-box testers ideas for test cases to apply when they receive the software for testing.

4. They may not necessarily understand the details of the code, but by listening to the review comments they can identify feature areas that sound troublesome or bug-prone.

2. Formal Review (Question: Explain formal review in static white box testing = 4 marks)

1. A formal review is the process under which static white-box testing is performed.

2. A formal review can range from a simple meeting between two programmers to a detailed, rigorous inspection of the code.

There are four essential elements to a formal review

1. Identify Problems: - .The goal of the review is to find problems with the software—not just items that are wrong, but missing items as well. All criticism should be directed at the code, not the person who created it. Participants shouldn't take any criticism personally. Leave your egos, emotions, and sensitive feelings at the door.

2. Follow Rules: - A fixed set of rules should be followed. They may set the amount of code to be reviewed (usually a couple hundred lines), how much time will be spent (a couple hours), what can be commented on, and so on. This is important so that the participants know what their roles are and what they should expect. It helps the review run more smoothly.

3. Prepare: - Each participant is expected to prepare for and contribute to the review. Depending on the type of review, participants may have different roles. They need to know what their duties and responsibilities are and be ready to actively fulfil them at the review. Most of the problems found through the review process are found during preparation, not at the actual review.

4. Write a Report: - The review group must produce a written report summarizing the results of the review and make that report available to the rest of the product development team. It's imperative that others are told the results of the meeting—how many problems were found, where they were found, and so on.

3. Peer Reviews (Question: Explain the concept of peer review = 4 marks)

1. The easiest way to get team members together and doing their first formal reviews of the software is through peer reviews, the least formal method.

2. Sometimes called buddy reviews, this method is really more of a discussion.

3. Peer reviews are often held with just the programmer who wrote the code and one or two other programmers or testers acting as reviewers.

4. Small group simply reviews the code together and looks for problems and oversights.

5. To assure that the review is highly effective all the participants need to make sure that the four key elements of a formal review are in place: Look for problems, follow rules, prepare for the review, and write a report.

6. As peer reviews are informal, these elements are often scaled back. Still, just getting together to discuss the code can find bugs.

4. Walkthroughs (Question: Explain the concept of peer review = 4 marks)

1. Walkthroughs are the next step up in formality from peer reviews.

2. In a walkthrough, the programmer who wrote the code formally presents (walks through) it to a small group of five or so other programmers and testers.

3. The reviewers should receive copies of the software in advance of the review so they can examine it and write comments and questions that they want to ask at the review.

4. Having at least one senior programmer as a reviewer is very important.

5. Inspections (Question: Explain the concept of peer review = 4 marks)

1. Inspections are the most formal type of reviews.

2. They are highly structured and require training for each participant.

3. Inspections are different from peer reviews and walkthroughs in that the person

who presents the code, the presenter or reader, isn't the original programmer.

4. These forces someone else to learn and understand the material being presented, potentially giving a different slant and interpretation at the inspection meeting.

5. The other participants are called inspectors.

6. Each is tasked with reviewing the code from a different perspective, such as a user, a tester, or a product support person.

7. This helps bring different views of the product under review and very often identifies different bugs.

8. One inspector is even tasked with reviewing the code backward—that is, from the end to the beginning—to make sure that the material is covered evenly and completely.

6. Coding Standards and Guidelines (Question: Explain coding standards and guidelines = 4 marks)

1. The code may operate properly but may not be written to meet a specific standard or guideline.

2. Its equivalent to writing words that can be understood and get a point across but don't meet the grammatical and syntactical rules of the English language.

3. Standards are the established, fixed, have-to-follow-the rules—the do's and don'ts.

4. Guidelines are the suggested best practices, the recommendations, the preferred way of doing things.

5. Standards have no exceptions, short of a structured waiver process. Guidelines can be a bit loose.

There are three reasons for adherence to a standard or guideline:

1. Reliability: - It's been shown that code written to a specific standard or guideline is more reliable and bug-free than code that isn't.

2. Readability/Maintainability: - Code that follows set standards and guidelines is easier to read, understand, and maintain.

3. Portability: - Code often has to run on different hardware or be compiled with different compilers.

7. Generic Code Review Checklist (Question: Explain the generic code review checklist = 4 marks) Data Reference Errors

Data reference errors are bugs caused by using a variable, constant, array, string, or record that hasn't been properly initialized for how it's being used and referenced.

1. Is an uninitialized variable referenced? Looking for omissions is just as important as looking for errors.

2. Are array and string subscripts integer values and are they always within the bounds of the array's or string's dimension?

3. Are there any potential "off by one" errors in indexing operations or subscript references to arrays?

4. Is a variable used where a constant would actually work better—for example, when checking the boundary of an array?

5. Is a variable ever assigned a value that's of a different type than the variable? For example, does the code accidentally assign a floating-point number to an integer variable?

6. Is memory allocated for referenced pointers?

7. If a data structure is referenced in multiple functions or subroutines, is the structure defined identically in each one?

8. Data Declaration Errors (Question: Explain data declaration error and computational errors – 4 marks each = 8 marks)

Data declaration bugs are caused by improperly declaring or using variables or constants.

1. Are all the variables assigned the correct length, type, and storage class? For example, should a variable be declared as a string instead of an array of characters?

2. If a variable is initialized at the same time as it's declared, is it properly initialized and consistent with its type?

3. Are there any variables with similar names? This isn't necessarily a bug, but it could be a sign that the names have been confused with those from somewhere else in the program.

4. Are any variables declared that are never referenced or are referenced only once?

5. Are all the variables explicitly declared within their specific module? If not, is it understood that the variable is shared with the next higher module?

9. **Computation Errors**

Computational or calculation errors are essentially bad math. The calculations don't result in the expected result.

1. Do any calculations that use variables have different data types, such as adding an integer to a floating-point number?

2. Do any calculations that use variables have the same data type but are different lengths—adding a byte to a word, for example?

3. Are the compiler's conversion rules for variables of inconsistent type or length understood and taken into account in any calculations?

4. Is the target variable of an assignment smaller than the right-hand expression?

5. Is overflow or underflow in the middle of a numeric calculation possible?

6. Is it ever possible for a divisor/modulus to be zero?

7. For cases of integer arithmetic, does the code handle that some calculations, particularly division, will result in loss of precision?

8. Can a variable's value go outside its meaningful range? For example, could the result of a probability be less than 0% or greater than 100%?

9. For expressions containing multiple operators, is there any confusion about the order of evaluation and is operator precedence correct? Are parentheses needed for clarification?

10. Comparison Errors

Comparison and decision errors are very susceptible to boundary condition problems.

1. Are the comparisons correct? It may sound pretty simple, but there's always confusion over whether a comparison should be less than or less than or equal to.

2. Are there comparisons between fractional or floating-point values? If so, will any precision problems affect their comparison? Is 1.00000001 close enough to 1.00000002 to be equal?

3. Does each Boolean expression state what it should state? Does the Boolean calculation work as expected? Is there any doubt about the order of evaluation?

4. Are the operands of a Boolean operator Boolean? For example, is an integer variable containing integer values being used in a Boolean calculation?

11. Control Flow Errors

Control flow errors are the result of loops and other control constructs in the language not behaving as expected.

1. They are usually caused, directly or indirectly, by computational or comparison errors.

2. If the language contains statement groups such as begin...end and do...while, are the ends explicit and do they match their appropriate groups?

3. Will the program, module, subroutine, or loop eventually terminate? If it won't, is that acceptable?

4. Is there a possibility of premature loop exit?

5. Is it possible that a loop never executes? Is it acceptable if it doesn't?

6. If the program contains a multi way branch such as a switch...case statement, can the index variable ever exceed the number of branch possibilities?

12. Subroutine Parameter Errors

Subroutine parameter errors are due to incorrect passing of data to and from software subroutines. 1. Do the types and sizes of parameters received by a subroutine match those sent by the calling code? Is the order correct?

2. If a subroutine has multiple entry points (yuck), is a parameter ever referenced that isn't associated with the current point of entry?

3. If constants are ever passed as arguments, are they accidentally changed in the subroutine?

4. Does a subroutine alter a parameter that's intended only as an input value?

5. Do the units of each parameter match the units of each corresponding argument—English versus metric, for example?

6. If global variables are present, do they have similar definitions and attributes in all referencing subroutines?

13. Input / Output Errors

These errors include anything related to reading from a file, accepting input from a keyboard or mouse, and writing to an output device such as a printer or screen. The items presented here are very simplified and generic.

1. Does the software strictly adhere to the specified format of the data being read or written by the external device?

2. If the file or peripheral isn't present or ready, is that error condition handled?

3. Does the software handle the situation of the external device being disconnected, not available, or full during a read or write?

4. Are all conceivable errors handled by the software in an expected way?

5. Have all error messages been checked for correctness, appropriateness, grammar, and spelling?

II. Dynamic White Box Testing

1. Dynamic White Box Testing (Question: Explain the concept of dynamic white box testing = 4 marks)

1. Dynamic white-box testing, in a nutshell, is using information you gain from seeing what the code does and how it works to determine what to test, what not to test, and how to approach the testing. 2. Another name commonly used for dynamic white-box testing is structural testing because you can see and use the underlying structure of the code to design and run your tests.

3. Dynamic white-box testing isn't limited just to seeing what the code does. It also can involve directly testing and controlling the software.

The four areas that dynamic white-box testing encompasses are

1. Directly testing low-level functions, procedures, subroutines, or libraries. In Microsoft Windows, these are called Application Programming Interfaces (APIs).

2. Testing the software at the top level, as a completed program, but adjusting your test cases based on what you know about the software's operation.

3. Gaining access to read variables and state information from the software to help you determine whether your tests are doing what you thought and, being able to force the software to do things that would be difficult if you tested it normally.

4. Measuring how much of the code and specifically what code you "hit" when you run your tests and then adjusting your tests to remove redundant test cases and add missing ones.

2. Dynamic White Box Testing versus Debugging (Question: differentiate between dynamic white box testing and debugging = 4 marks)

1. It's important not to confuse dynamic white-box testing with debugging.

2. The goal of dynamic white-box testing is to find bugs. The goal of debugging is to fix them.

3. They do overlap, however, in the area of isolating where and why the bug occurs. You'll learn more about this later about, "Reporting What You Find," but for now, think of the overlap this way.

4. As a software tester, you should narrow down the problem to the simplest test case that demonstrates the bug.

5. If its white-box testing, that could even include information about what lines of code look suspicious.

6. The programmer who does the debugging picks the process up from there, determines exactly what is causing the bug, and attempts to fix it.

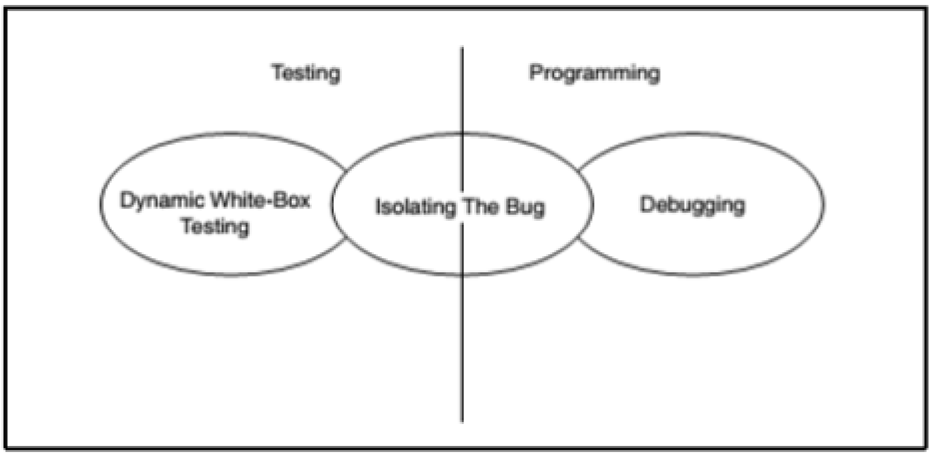

Figure 1 3.

3. Testing the Pieces

1. The big-bang model was the easiest but the most chaotic.

2. Everything was put together at once and, with fingers crossed, the team hoped that it all worked and that a product would be born.

3. Testing in such a model would be very difficult.

4. In dynamic black-box testing, taking the product in one entire block and exploring it to see errors.

- Unit and Integration Testing

    1. As a large project or an application software is divided into small modules

or units to reduce the complexity and to minimize the failure rate of the software.

2. Unit testing tests all the modules separately for the functionality and the the correctness of each module.

3. The integration testing is than implemented for the following reasons

- To test that the output of one unit or module when given as an input to the second module does not affect the output and the correctness of the integrated module.
- The change made to one module gives the effective and correct output of the integrated module and software as a product does not fail.
- The parameters of one module match the parameters of the other module with respect to the permissible values, boundary conditions, correctness and utilization.
- The figure shows the concept of unit testing and integration testing in V model.
- There are two basic types of integration (i) top down integration (ii) bottom up integration.

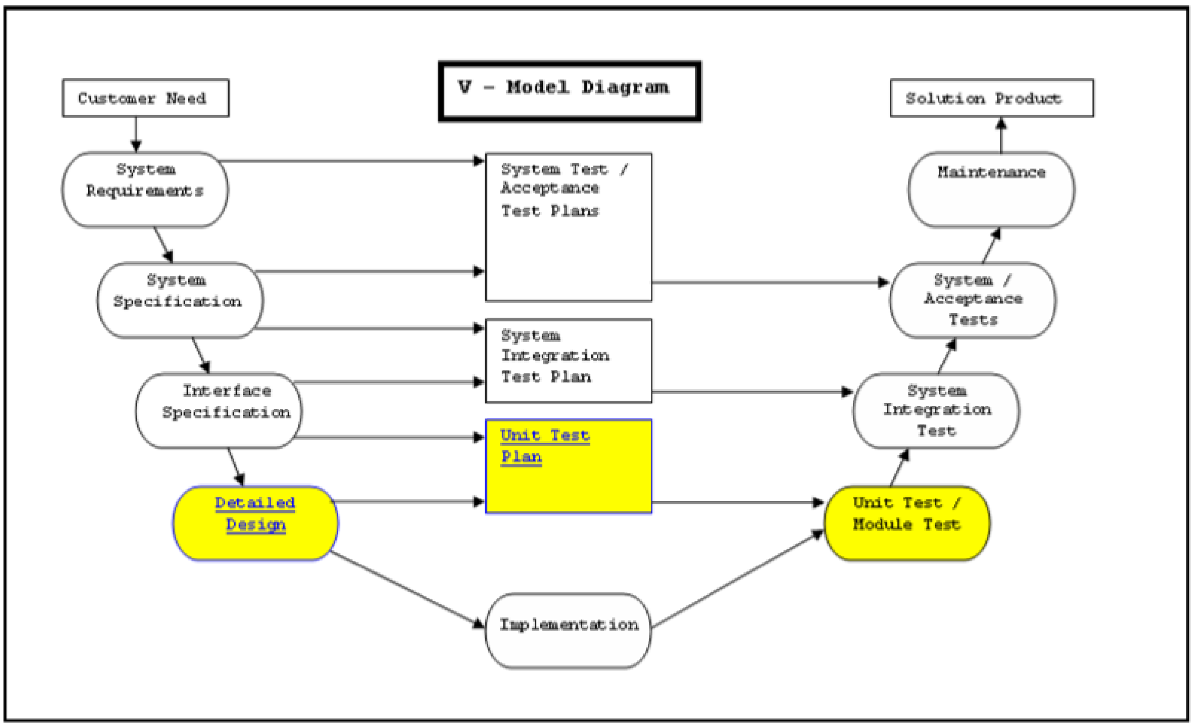

Figure 2

4. Data Coverage

1. The logical approach is to divide the code just as you did in black-box testing into its data and its states (or program flow).

2. By looking at the software from the same perspective, you can more easily map the white-box information you gain to the black-box cases you've already written.

3. Consider the data first. Data includes all the variables, constants, arrays, data structures, keyboard and mouse input, files and screen input and output, and I/O to other devices such as modems, networks, and so on.

For example

```
#include<stdio.h>
void main()
{
int i , fact= 1, n;
printf("enter the number ");
scanf("%d",&n);
for(i =1 ;i <=n;i++)
fact = fact * i;
printf ("the factorial of a number is "%d", fact);
}
```

The declaration of data is complete with the assignment statement and the variable declaration statements. All the variable declared are properly utilized.

- **Data Flow**

  1. Data flow coverage involves tracking a piece of data completely through the software.

  2. At the unit test level this would just be through an individual module or function.

  3. The same tracking could be done through several integrated modules or even through the entire software product—although it would be more time consuming to do so.

4. During data flow , the check is made for the proper declaration of variables declared and the loops used are declared and used properly.

For example

```
#include<stdio.h>
void main()
{
int i , fact= 1, n;
printf(“enter the number “);
scanf(“%d”,&n);
for(i =1 ;i <=n;i++)
fact = fact * i;
printf (“the factorial of a number is ”%d”, fact);
}
```

- **Sub-Boundaries**

 1. These are probably the most common examples of sub-boundaries that can cause bugs, but every piece of software will have its own unique sub boundaries, too.

 For examples

1. A module that computes taxes might switch from using a data table to using a formula at a certain financial cut-off point.

2. An operating system running low on RAM may start moving data to temporary storage on the hard drive.

3. This sub-boundary may not even be fixed. It may change depending on how much space remains on the disk.

- **Formula and Equations**

1. Very often, formulas and equations are buried deep in the code and their presence or effect isn't always obvious from the outside.

2. A financial program that computes compound interest will definitely have this formula somewhere in the software:

$A=P(1+r/n) nt$

where P = principal amount r = annual interest rate n = number of times the interest is compounded per year t = number of years A = amount after time t

3. A good black-box tester would hopefully choose a test case of n=0, but a whitebox tester, after seeing the formula in the code, would know to try n=0 because that would cause the formula to blow up with a divide-by-zero error.

- **Error Forcing**

1. The last type of data testing covered in this chapter is error forcing.

2. Testing the software in a debugger has the ability to watch variables and see what values they hold.

3. In the preceding compound interest calculation, if you couldn't find a direct way to set the number of compounding (n) to zero, you could use your debugger to force it to zero.

4. The software would then have to handle it or not

5. Code Coverage

1. Black-box testing, testing the data is only half the battle.

2. For comprehensive coverage you must also test the program's states and the program's flow among them.

3. You must attempt to enter and exit every module, execute every line of code, and follow every logic and decision path through the software.

4. Examining the software at this level of detail is called Code-coverage analysis is a dynamic white-box testing technique because it requires you to have full access to the code to view what parts of the software you pass through when you run your test cases.

- **Program Statements and Line Coverage**

1. The most straightforward form of code coverage is called statement coverage or line coverage.

2. If you're monitoring statement coverage while you test your software, your goal is to make sure that you execute every statement in the program at least once.

3. With line coverage the tester tests the code line by line giving the relevant output.

For example

```
#include<stdio.h>
void main()
{
int i , fact= 1, n;
printf("enter the number ");
scanf("%d", &n);
for(i =1 ;i <=n; i++)
fact = fact * i;
printf ("the factorial of a number is "%d", fact);
}
```

- **Branch Coverage**

1. Attempting to cover all the paths in the software is called path testing.

2. The simplest form of path testing is called branch coverage testing.

3. To check all the possibilities of the boundary and the sub boundary conditions and it's branching on those values.

4. Test coverage criteria requires enough test cases such that each condition in a decision takes on all possible outcomes at least once, and each point of entry to a program or subroutine is invoked at least once.

5. Every branch (decision) taken each way, true and false.

6. It helps in validating all the branches in the code making sure that no branch leads to abnormal behaviour of the application.

- **Condition Coverage**

1. Just when you thought you had it all figured out, there's yet another complication to path testing. 2. Condition coverage testing takes the extra conditions on the branch statements into account.