Important Instructions to examiners:

1) The answers should be examined by key words and not as word-to-word as given in themodel answer scheme.
2) The model answer and the answer written by candidate may vary but the examiner may tryto assess the understanding level of the candidate.
3) The language errors such as grammatical, spelling errors should not be given moreImportance (Not applicable for subject English and Communication Skills).
4) While assessing figures, examiner may give credit for principal components indicated in thefigure. The figures drawn by candidate and model answer may vary. The examiner may givecredit for anyequivalent figure drawn.
5) Credits may be given step wise for numerical problems. In some cases, the assumed constantvalues may vary and there may be some difference in the candidate's answers and model answer.
6) In case of some questions credit may be given by judgement on part of examiner of relevantanswer based on candidate's understanding.
7) For programming language papers, credit may be given to any other program based on equivalentconcept.

1. a) Attempt any three of the following : Marks 12

a) List eight traits (properties) of good software tester.

(Each point- 1/2 Mark)

Ans: The eight traits (properties) of good software tester are: (explanation optional)

i. **Explorers**: Software testers aren't afraid to understand situations. They are very exploring to test new piece of software and make it error free.

ii. **Troubleshooters**: Software testers are good trouble shooters, they find out why something does not work.

iii. **Relentless**: Software testers always try to find bugs.

iv. **Creative**: Software tester's job is to be creative and even off-the-wall approaches to find new bugs.

v. **Perfectionists**: Software testers are always strives for perfection. But Software testers known when software creating problems and when it works properly.

vi. **Good judgment**: Software testers need to make decisions about what they will test, what inputs are using, when to stop testing, and how to break the code and find new bugs.

vii. **Tactful and diplomatic**: Software testers always bearer of bad news. They have to tell the programmers that their given functionality is not working.

viii. They are **persuasive**: When tester find the work, tester need to clear their point, demonstrating why the bug does indeed to fixed the following through on marketing it happens.

1. a) b) Describe spiral model in SDLC.

(Diagram - 2 Mark, Explanation - 2 Marks)

Ans: The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: **Planning, Risk Analysis, Engineering and Evaluation.** A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spirals, starting in the planning phase, requirements are gathered and risk is assessed. Each subsequent spiral builds on the baseline spiral.

Each time around the spiral involves six steps:

1. Determine objectives, alternatives, and constraints.

2. Identify and resolve risks.

3. Evaluate alternatives.

4. Develop and test the current level.

5. Plan the next level.

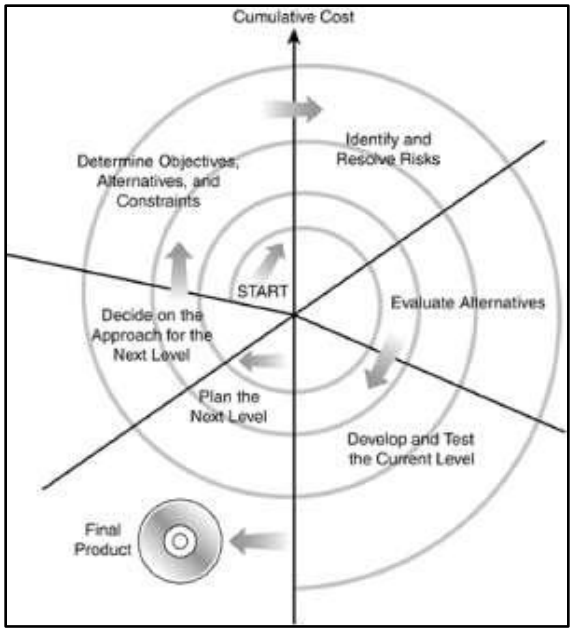
6. Decide on the approach for the next level.

Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase.

Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

In the spiral model, the angular component represents progress, and the radius of the spiral represents cost.

Diagram:



1. a) c) Explain performing a High level review of the specification.
(Suitable Meaning-1 Mark, Each sub point-1 Mark)

Ans: Defining a software product is a difficult process. The spec must deal with many unknowns, take a multitude of changing inputs, and attempt to pull them all together into a document that describes a new product. The process is an inexact science and is prone to having problems. So we must perform a high level review to remove all bugs in specification.

1. Pretend to be the Customer

The easiest thing for a tester to do when he first receives a specification for review is to pretend to be the customer. Do some research about who the customers will be? Talk to your marketing or sales people to get hints on what they know about the end user. If the product is an internal software project, find out who will be using it and talk to them.

It's important to understand the customer's expectations. Remember that the definition of quality means **"meeting the customer's needs."** As a tester, you must understand those needs to test that the software meets them.

Above all else, assume nothing. If you review a portion of the specification and don't understand it, don't assume that it's correct and go on. Eventually, you'll have to use this specification to design your software tests, so you'll eventually have to understand it. There's no better time to learn than now. If you find bugs along the way (and you will), all the better.

2. Research Existing Standards and Guidelines

Here are several examples of standards and guidelines to consider. This list isn't definitive. You should research what might apply to your software:

- **Corporate Terminology and Conventions.** If this software is tailored for a specific company, it should adopt the common terms and conventions used by the employees of that company.
- **Industry Requirements.** The medical, pharmaceutical, industrial, and financial industries have very strict standards that their software must follow.
- **Government Standards.** The government, especially the military, has strict standards.
- **Graphical User Interface (GUI).** If your software runs under Microsoft Windows or Apple Macintosh operating systems, there are published standards and guidelines for how the software should look and feel to a user.
- **Hardware and Networking Standards.** Low-level software and hardware interface standards must be adhered to, to assure compatibility across systems.

3. Review and Test Similar Software

Some things to look for when reviewing competitive products include

- **Scale.** Will your software be smaller or larger? Will that size make a difference in your testing?
- **Complexity.** Will your software be more or less complex? Will this impact your testing?
- **Testability.** Will you have the resources, time, and expertise to test software such as this?
- **Quality/Reliability.** Is this software representative of the overall quality planned for your software? Will your software be more or less reliable?

1. a) d) Explain the white –box testing with example.

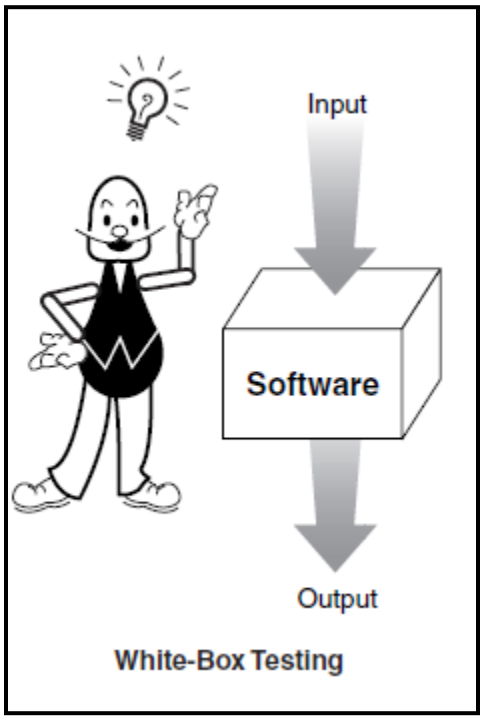
(Advantages and disadvantages are optional)

(Explanation - 2 Marks, Any example- 2 Marks)

Ans: **White Box Testing** (*also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing)* is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.

This method is named so because the software program, in the eyes of the tester, is like a white/transparent box; inside which one clearly sees.

Diagram:



In white-box testing, the software tester has access to the program's code and can examine it for clues to help him with his testing—he can see inside the box. Based on what he sees, the

tester may determine that certain numbers are more or less likely to fail and can tailor his testing based on that information.

EXAMPLE

A tester, usually a developer as well, studies the implementation code of a certain field on a webpage, determines all legal (valid and invalid) AND illegal inputs and verifies the outputs against the expected outcomes, which is also determined by studying the implementation code.

WHITE BOX TESTING ADVANTAGES

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.

WHITE BOX TESTING DISADVANTAGES

- Since tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.
- Since this method of testing it closely tied with the application being testing, tools to cater to every kind of implementation/platform may not be readily available.
- White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.

1. b) Attempt any one of the following : Marks 6

a) Explain the Code and –Fix model.

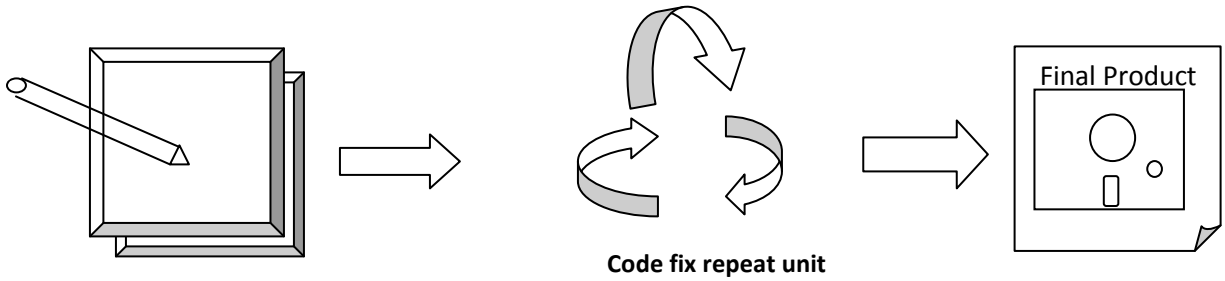
(Diagram -2 Mark, Explanation- 4 Marks)

Ans: Code and Fix Model:

The code and Fix model is as shown in fig is usually the one that project teams fall in to by default if they don't consciously attempt to use something else. It's a step up, procedurally from Big Bang model.

In this model in that at least requires some idea of what product requirement are.

Diagram:



This model plays a significant role between coding & fixing. A team using approach usually starts with rough idea of what they exactly want, does some sample design & then produced into long repeating cycle of coding, testing & fixing bugs. Code & fix has been used on many large & well known software products.

Example: Code & Fix model is used for Word processor or spreadsheet software

This model play very significant role in coding & fixing.

As a tester on a code-and-fix project, you need to be aware that you, along with the programmers, will be in a constant state of cycling. As often as every day you'll be given new or updated releases of the software and will set off to test it. You'll run your tests, report the bugs, and then get a new software release. You may not have finished testing the previous release when the new one arrives, and the new one may have new or changed features. Eventually, you'll get a chance to test most of the features, find fewer and fewer bugs, and then someone (or the schedule) will decide that it's time to release the product.

You will most likely encounter the code-and-fix model during your work as a software tester. It's a good introduction to software development and will help you appreciate the more formal methods.

1. b) b) State the types of test tools and explain any three.

(Type - 1 ½ Marks each, Any three type explanation- 1 ½ Marks)

Ans: The different types of test tools are as follows:

- Viewers and Monitors
- Drivers

- Stubs
- Stress and Load Tools
- Interference Injectors and Noise Generators
- Analysis Tools

Viewers and Monitors: A viewer or monitor test tool allows you to see details of the software's operation that you wouldn't normally be able to see. **Code coverage analyzers provide a means for you to see what lines of code are executed, what functions are run, and what code paths are followed when you run your tests**. A code coverage analyzer is an example of a viewing tool. Most code coverage analyzers are invasive tools because they need to be compiled and linked into the program to access the information they provide**. The code debuggers that come with most compilers are also considered viewers because they allow programmers or white-box testers to view internal variable values and program states. Anything that lets you see into the system and look at data that the average user wouldn't be able to see can be classified as a viewer test tool.**

Drivers: Drivers are tools used to control and operate the software being tested. One of the simplest examples of a driver is a batch file, a simple list of programs or commands that are executed sequentially. In the days of MS-DOS, this was a popular means for testers to execute their test programs. They'd create a batch file containing the names of their test programs, start the batch running, and go home. With today's operating systems and programming languages, there are much more sophisticated methods for executing test programs. For example, a complex Perl script can take the place of an old MS-DOS batch file, and the Windows Task Scheduler can execute various test programs at certain times throughout the day.

Stubs: Stubs are essentially the opposite of drivers in that they don't control or operate the software being tested; they instead receive or respond to data that the software sends. If you're testing software that sends data to a printer, one way to test it is to enter data, print it, and look at the resulting paper printout. That would work, but it's fairly slow, inefficient, and error prone. Could you tell if the output had a single missing pixel or if it was just slightly off in color? If you instead replaced the printer with another computer that

was running stub software that could read and interpret the printer data, it could much more quickly and accurately check the test results. **Stubs are frequently used when software needs to communicate with external devices. Often during development these devices aren't available or are scarce. A stub allows testing to occur despite not having the hardware and it makes testing more efficient**.

Stress and Load Tools: Stress and load tools induce stress and loads to the software being tested. A word processor running as the only application on the system, with all available memory and disk space, probably works just fine. But, if the system runs low on these resources, you'd expect a greater potential for bugs. You could copy files to fill up the disk, run lots of programs to eat up memory, and so on, but these methods are inefficient and non-exact. **A stress tool specifically designed for this would make testing much easier. Load tools are similar to stress tools in that they create situations for your software that might otherwise be difficult to create**. For example, commercially available programs can be run on Web servers to load them down by simulating a set number of connections and hits. You might want to check those 10,000 simultaneous users and 1 million hits a day can be handled without slowing response times. With a load tool, you can simply dial in that level, run your tests, and see what happens.

Interference Injectors and Noise Generators: Another class of tools is interference injectors and noise generators. They're similar to stress and load tools but are more random in what they do. The Stress tool, for example, has an executor mode that randomly changes the available resources. A program might run fine with lots of memory and might handle low memory situations, but it could have problems if the amount of available memory is constantly changing. The executor mode of stress would uncover these types of bugs.

When deciding where and how to use interference injectors and noise generators, think about what external influences affect the software you're testing, and then figure out ways to vary and manipulate those influences to see how the software handles it.

Analysis Tools: Most software testers use the following common tools to make their everyday jobs easier. They're not necessarily as fancy as the tools discussed so far. They're **often taken for granted, but they get the job done and can save you a great deal of time.**

- **Word processing software**
- **Spreadsheet software**
- **Database software**
- **File comparison software**
- **Screen capture and comparison software**
- **Debugger**
- **Binary-hex calculator**
- **Stopwatch**
- **VCR or camera**

2. Attempt any four of the following : Marks 16

a) Explain equivalence partitioning with example.

(Explanation - 2 Marks, Any example - 2 Marks)

Ans: Selecting test cases is the single most important task that software testers do and equivalence partitioning, sometimes called equivalence classing, is the means by which they do it.

Equivalence partitioning is a software testing technique to minimize number of permutation and combinations of input data. In Equivalence partitioning, data is selected in such a way that it gives as many different outputs as possible with the minimal set of data. The process of methodically reducing the huge (infinite) set of possible test cases, into a much smaller but still equally effective set. There are two steps that we need to follow if you want to use Equivalence partitioning in your project:

i. **Identify Equivalence classes of partition.**
ii. **Picking one value from each partition for the complete coverage.**

The main advantage of this technique is that testing efforts are minimized and at that the same time coverage is also ensured.

Example: Consider a very simple function for awarding grades to the students; this program follows this guideline to award grades:

Marks 00 – 39 ---------- Grade D

Marks 40 – 59 ---------- Grade C

Marks 60 – 70 ---------- Grade B

Marks 71 – 100 ---------- Grade A

Based on the equivalence portioning techniques, partitions for this program could be as follows:

Marks between 0 to 39 – Valid Input

Marks between 40 to 59 – Valid Input

Marks between 60 to 70 – Valid Input

Marks between 71 to 100 – Valid Input

Marks less than 0 – Invalid Input

Marks more than 100 – Invalid Input

Non-numeric Input – Invalid Input

2. b) Describe the low- level specification test techniques.

(Specification Attributes Checklist: 2 Marks - any two points, Specification Terminology Checklist: 2 Marks - any two points)

Ans: After you complete the high-level review of the product specification, you'll have a better Understanding of what your product is and what external influences affect its design. Armed With this information, you can move on to testing the specification at a lower level.

Specification Attributes Checklist:

A good, well-thought-out product specification, with "all its t's crossed and its i's dotted," has eight important attributes:

• **Complete.** Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?

• **Accurate.** Is the proposed solution correct? Does it properly define the goal? Are there any errors?

• **Precise, Unambiguous, and Clear.** Is the description exact and not vague? Is there a

single interpretation? Is it easy to read and understandable?

• **Consistent.** Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?

• **Relevant.** Is the statement necessary to specify the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?

• **Feasible.** Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?

• **Code-free.** Does the specification stick with defining the product and not the underlying software design, architecture, and code?

• **Testable.** Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

Specification Terminology Checklist:

A complement to the previous attributes list is a list of problem words to look for while reviewing a specification. Specification Terminology Checklist is a list of problematic words. Appearance of these words often signifies that a feature is an yet completely throughout. This specification may go onto clarify or elaborate on the, or it may leave them ambiguous. These words are:

Always, Every, All, None, Never ; If you see words such as these that denote something as certain or absolute, make sure that it is, indeed, certain. Put on your tester's hat and think of cases that violate them.

• **Certainly, Therefore, Clearly, Obviously, Evidently ;** These words tend to persuade you into accepting something as a given. Don't fall into the trap.

• **Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly.** These words are too vague. It's impossible to test a feature that operates "sometimes."

• **Etc., And So Forth, And So On, Such As.** Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.

• **Good, Fast, Cheap, Efficient, Small, Stable.** These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.

• **Handled, Processed, Rejected, Skipped, Eliminated.** These terms can hide large amounts of functionality that need to be specified.

• **If…Then…(but missing Else).** Look for statements that have "If…Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen.

2. c) Describe a four important traits common to good user interface.
(Any four traits - 1 Mark each)

Ans: Following are the important traits common to a good user interface:

i. **Follows standard or Guidelines:** The single most important user interface trait is that your software follows existing standards and guidelines—or has a really good reason not to. If your software is running on an existing platform such as Mac or Windows, the standards are set. Each book goes into meticulous detail about how software that runs on each platform should *look* and *feel* to the user.

ii. **Intuitive**: When you are testing a user interface, consider the following things and how they might apply to gauging how intuitive your software is:

- **Is the user interface clean, unobtrusive, not busy?** The UI shouldn't get in the way of what you want to do. The functions you need or the response you're looking for should be obvious and be there when you expect them.
- **Is the UI organized and laid out well?** Does it allow you to easily get from one function to another? Is what to do next obvious? At any point can you decide to do nothing or even back up or back out? Are your inputs acknowledged? Do the menus or windows go too deep?
- **Is there excessive functionality?** Does the software attempt to do too much, either as a whole or in part? Do too many features complicate your work? Do you feel like you're getting information overload?
- **If all else fails, does the help system really help you?**

iii. **Consistent:** Consistency within your software and with other software is a key attribute. Users develop habits and expect that if they do something a certain way in one program, another will do the same operation the same way. An example of how two Windows applications, which should be following a standard, aren't consistent. In Notepad, Find is

accessed through the Search menu or by pressing F3. In WordPad, a very similar program, it's accessed through the Edit menu or by pressing Ctrl+F.

Think about a few basic items as you review your product:

- **Shortcut keys and menu selections**
- **Terminology and naming.**
- **Audience.**
- **Placement and keyboard equivalents for buttons.**

iv. **Flexible: Users like choices**—not too many, but enough to allow them to select what they want to do and how they want to do it. The Windows Calculator has two views: Standard and Scientific. Users can decide which one they need for their task or the one they're most comfortable using. Of course, with flexibility comes complexity. In the Calculator example you'll have a much larger test effort than if there's just one view.

- State Jumping
- State Termination and Skipping
- Data Input and Output

v. **Comfortable:** Software should be comfortable to use. It shouldn't get in the way or make it difficult for a user to do his work. Software comfort is a pretty touchy-feely concept. Researchers have spent their careers trying to find the right formula to make software comfortable. It can be a difficult concept to quantify, but you can look for a few things that will give you a better idea of how to identify good and bad software comfort:

- Appropriateness.
- Error handling.
- Performance.

vi. **Correct:** The comfort trait is admittedly a bit fuzzy and often can be left to interpretation. Correctness, though, isn't. When you're testing for correctness, you're testing whether the UI does what it's supposed to do. Correctness problems such as this are usually obvious and will be found in your normal course of testing against the product specification. You should pay attention to some areas in particular, however:

- Marketing differences
- Language and spelling

- Bad media
- WYSIWYG (what you see is what you get).

vii. **Useful:** The final trait of a good user interface is whether it's useful. Remember, you're not concerned with whether the software itself is useful, just whether the particular feature is. A popular term used in the software industry to describe unnecessary or gratuitous features is dancing bologna. It doesn't matter whether the dancing bologna is in a solitaire program or a heart monitor machine; it's bad for the user and means extra testing for you. When you're reviewing the product specification, preparing to test, or actually performing your testing, ask yourself if the features you see actually contribute to the software's value. Do they help users do what the software is intended to do? If you don't think they're necessary, do some research to find out why they're in the software?

2. d) What is Beta testing?

(Definition- 1 Mark, Explanation-3 Marks)

Ans: Beta testing is the term used to describe the external testing process in which the software is sent out to a select group of potential customers who use it in a real-world environment. Beta testing usually occurs toward the end of the product development cycle and ideally should just be a validation that the software is ready to release to real customers.

The goals of a beta test can vary considerably from getting the press to write early reviews of the software to user interface validation to a last-ditch effort in finding bugs. As a tester, you need to make it known to the person managing the beta testing what, if anything, you want to achieve from it.

From a test standpoint, there are several things to think about when planning for or relying on a beta test:

- **Who are the beta testers?** Since a beta test can have different goals, it's important to understand who the beta participants are. For example, you may be interested in identifying any remaining usability bugs left in the software, but the beta testers may all be experienced techies who are more concerned with the low-level operation and not usability. If your area of the software is to be beta tested, make sure that you define what

types of beta testers you need in the program so that you can receive the most benefit out of it.

- Similarly, **how will you know if the beta testers even use the software?** If 1,000 beta testers had the software for a month and reported no problems, would that mean there were no bugs, that they saw bugs but didn't report them, or that the disks were lost in the mail? It's not uncommon for beta testers to let the software sit for days or weeks before they try to use it, and when they do, only use it for a limited time and a limited set of features. Make sure that you or someone running the beta program follows up with the participants to make sure that they're using the software and meeting the plan's goals.
- Usability testing is another area that beta testing can contribute to if the participants are well chosen—a good mix of experienced and inexperienced users. **They'll be seeing the software for the first time and will readily find anything that's confusing or difficult to use.**
- Besides configuration, compatibility, and usability, beta tests are surprisingly poor ways to find bugs. The participants often don't have a lot of time to use the software, so they won't find much more than superficial, obvious problems—ones that you likely already know about. And, because beta testing usually occurs near the end of the development cycle, there's not much time to fix bugs that are found.

2. e) Explain the random testing in detail.
(Definition -1 Mark, Explanation- 3 Marks)

Ans: **Monkey testing is a random testing performed by automated testing tools.** These automated testing tools are considered "Monkeys" if they were kept random. We call them "monkeys" because it is widely believed that if we allow six monkeys to pound on six type writers at random, for a million years, they will re-create all the works of Isaac Asimov.

There are "Smart Monkeys" and "Dumb Monkey". "Smart Monkeys" are valuable for load and stress testing, they will find a significant number of bugs, but are also very expensive to develop. "Dumb Monkeys", on the other hand are inexpensive to develop, are able to do some basic testing but they will find few bugs. However, the bugs "Dumb Monkeys" do find will be hangs and crashes i.e. the bugs you least want to have in your software product. Monkey testing can be valuable, but they should not be your only testing.

It's also a good idea to program your monkey to operate only on the software's you are testing. If it's randomly testing clicking all over the screen, it could click the exit command and stop the program. Since the monkey doesn't know that the program closed and it will keep on going. **Another good feature to make your monkey semi conductor is crash recognition .**

2. f) List any four computation errors.
(Any four points-1 Mark each)

Ans: Computational or calculation errors are essentially bad math. The calculations don't result in the expected result. The types of computation errors are as follows:

i. Do any calculations that use variables have different data types, such as adding an integer to a floating-point number?
ii. Do any calculations that use variables have the same data type but are different lengths—adding a byte to a word, for example?
iii. Are the compiler's conversion rules for variables of inconsistent type or length understood and taken into account in any calculations?
iv. Is the target variable of an assignment smaller than the right-hand expression?
v. Is overflow or underflow in the middle of a numeric calculation possible?
vi. Is it ever possible for a divisor/modulus to be zero?
vii. For cases of integer arithmetic, does the code handle that some calculations, particularly division, will result in loss of precision?
viii. 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%?
ix. For expressions containing multiple operators, is there any confusion about the order of evaluation and is operator precedence correct? Are parentheses needed for clarification?

3. Attempt any four of the following : Marks 16

a) **Explain the data coverage in detail.**

(Data flow-1 Mark, sub boundaries-1 Mark, formulas and equations-1 Mark and error coverage-1 Mark)

Ans: Data Coverage

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. The data coverage covers following 4 major areas such as:

Data Flow

Data flow coverage involves tracking a piece of data completely through the software. If we test a function at this low level, we would use a debugger and watch variables to view the data as the program runs with black-box testing, we only know what the value of the variable is at the beginning and at the end. With dynamic white-box testing we could also check intermediate values during program execution. Based on what we see we might decide to change some of our test cases to make sure the variable takes on interesting or even risky interim values.

Sub-Boundaries

Every piece of software will have its own unique sub-boundaries.for examples:

- An operating system running low on RAM may start moving data to temporary storage on the hard drive. This sub-boundary may not even be fixed. It may change depending on how much space remains on the disk.
- To gain better precision, a complex numerical analysis program may switch to a different equation for solving the problem depending on the size of the number.

  If we perform white-box testing, we need to examine the code carefully to look for sub-boundary conditions and create test cases that will exercise them. Ask the programmer who wrote the code if he/she knows about any of these situations and pay special attention to internal tables of data because they're especially prone to sub-boundary conditions.

Formulas and Equations

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

A good black-box tester would hopefully choose a test case of n=0, but a white-box 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.

But, what if n was the result of another computation? Maybe the software sets the value of n based on other user input or algorithmically tries different n values in an attempt to find the lowest payment. we need to ask our self if there's any way that n can ever become zero and figure out what inputs to feed the program to make that happen.

Error Forcing

The last type of data coverage is error forcing. If we are running the software that we are testing in a debugger, we don't just have the ability to watch variables and see what values they hold we can also force them to specific values.

In the above compound interest calculation, if we couldn't find a direct way to set the number of compound (n) to zero, we could use our debugger to force it to zero. The software would then have to handle it or not.

If we take care in selecting our error forcing scenarios and double-check with the programmer to assure that they're valid, error forcing can be an effective tool.

3. b) What are the goals of test-case planning? Explain it.

(Each point-1 Mark)

Ans: The goals of test case planning are as given below:

1. **Organization:** Proper planning will organize test cases so that all testers & other project team members can review & use them effectively.
2. **Repeatability:** Same test cases run several times to look new bugs & to make sure that old ones have been fixed.
3. **Tracking:** Tracking of -how many test cases did you plan to run?
   -How many did you run on last software release?
   -How many passed & how many failed?
   -Were any test cases skipped?
4. **Proof of testing:** In a few high-risk industries, the software test team must prove that it did indeed run the tests that it planned to run. It could actually be illegal to release software in which a few test cases were skipped. Proper test case planning & tracking provides a means for proving what was tested.

3. c) Describe a boundary condition in detail.

(Explanation - 3 Marks, Example-1 Mark)

Ans:

Boundary conditions: Many systems have tendency to fail on boundary. So it is important to test boundary conditions of application. E.g. If we want to display numbers from 10 to 49, the condition will be, If (n>=10 AND n<50).

But if you have given the condition as: if(n>10 AND n<50) then 10 will not get printed. Or if (n>=10 AND n<=50) then 50 will also get printed. So for this example we have to test 9, 10, 11 and 48,49,50 values of _n'. These values are respectively lower_boundary-1, lower_boundary, lower_boundary+1, upper_boundary-1, upper_boundary, upper_boundary+1.

Types of boundary conditions: Numeric, character, position, quality, speed, location, size.

For this type the characteristics can be: First/Last, Min/Max, Over/Under, Highest/Lowest.

For each boundary condition include boundary value in at least one valid test case. Include value just beyond boundary in at least one invalid test case. Include test cases with input such that outputs at boundaries are produced.

Testing the Boundary Edges:

First-1/Last+1

Start-1/Finish+1

Largest+1/Smallest-1

Min-1/Max+1

Just Over/Just Under

E.g. If a text entry field allows 1 to 255 characters, try entering 1, 2 character and 254,255 characters as the valid partition. Enter 0 and 256 characters as the invalid partitions.

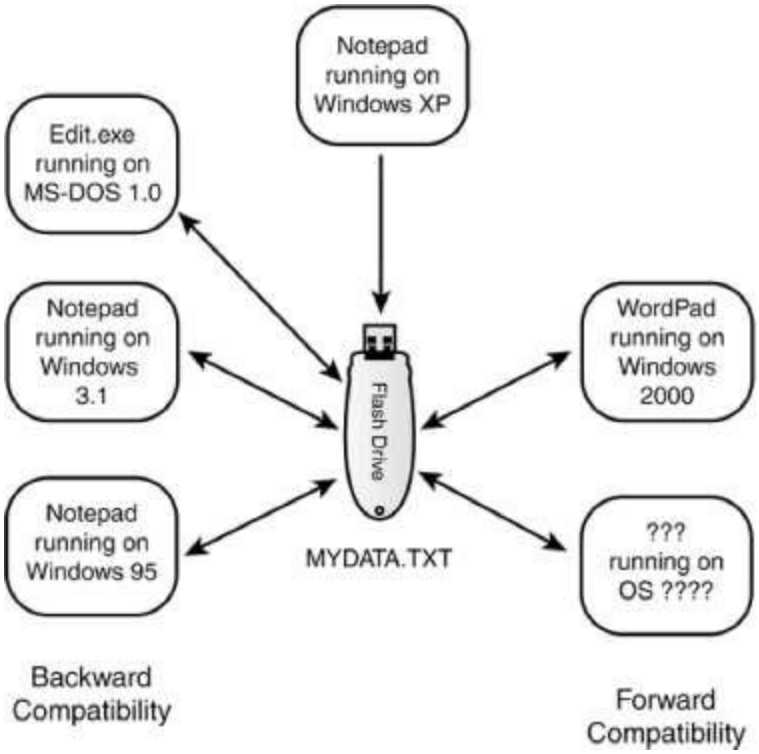
3. d) Explain the background and forward compatibility in detail.
(Note: Read background as backward)
(Explanation-3 Marks, Diagram -1 Mark)

Ans: Backward and Forward Compatibility

If something is backward compatible, it will work with previous versions of the software. If something is forward compatible, it will work with future versions of the software.

The simplest demonstration of backward and forward compatibility is with a .txt or text file. As shown in fig. a text file created using Notepad 98 running under Windows 98 is backward compatible all the way back to MS-DOS 1.0. It's also forward compatible to Windows XP service pack 2 and likely will be beyond that.

Figure: Backward and forward compatibility define what versions will work with your software or data files.



3. e) Describe gray box testing.

(Definition -1 Mark, Explanation- 2 Marks, Example -1 Mark (any other suitable example on gray box testing will do)

Ans: Gray-Box Testing

Gray box testing is the testing of software application using effective combination of both White box testing & Black box testing method. This is nice & powerful idea to test the application.

The white box testing means tester is aware of internal structure of code but the black box tester doesn't aware the internal structure of code.

In the **Gray box testing** tester usually has knowledge of limited access of code and based on this knowledge the test cases are designed and the software application under test treat as a black box & tester test the application from outside. Also the *Gray box testing* is not a black box testing method because the tester knows some part of the internal structure of code.

So Gray Box Testing approach is the testing approach used when some knowledge of internal structure but not in detailed.

The name comes because the application for tester is like a gray box like a transparent box and tester see inside it but not fully transparent & can see partially in it. As tester doesn't require the access to code the gray box testing is known as unbiased & non-intrusive.

To test the Web Services application usually the Gray box testing is used.

Gray Box Testing Example:

We will take example of web application testing. To explore gray testing will take a simple functionality of web application. You just want to enter email id as input in the web form & upon submitting the valid email id user & based on users interest (fields entered) user should get some articles over email. The validation of email is using Java Script on client side only. In this case if tester doesn't knows the internal structure of the implementations then you might test the web application of form with some cases like Valid Email Id, Invalid Email ID & based on this will check whether functionality is working or not.

But tester is aware of some internal structure & if system is making the assumptions like

- System will not get Invalid email ID
- System will not send email to invalid email ID
- System will not receive failure email notifications.

In this type of testing you have to test the application by disabling the Java Script, it might be possible due to any reason Java Script is failed & System get Invalid email to process & all assumptions made by system will failed, as a result incorrect inputs are send to system, so

- System will get Invalid email ID to process
- System will send email to invalid email ID
- System will receive failure email notifications.

4. a) Attempt any three of the following: Marks 12

a) Explain test phases, test strategy, test schedule and test cases.

(Each term-1 Mark)

Ans: **Test Phases**

To plan the test phases, the test team will look at the proposed development model and decide whether unique phases, or stages, of testing should be performed over the course of the project. In a code-and-fix model, there's probably only one test phase—test until someone yells stop. In the waterfall and spiral models, there can be several test phases from examining the product spec to acceptance testing. Yes, test planning is one of the test phases. The test planning process should identify each proposed test phase and make each phase known to the project team. This process often helps the entire team form and understands the overall development model.

Test Strategy

The test strategy describes the approach that the test team will use to test the software both overall and in each phase. If you were presented with a product to test, you'd need to decide if it's better to use black-box testing or white-box testing. If you decide to use a mix of both techniques, when will you apply each and to which parts of the software?

It might be a good idea to test some of the code manually and other code with tools and automation. If tools will be used, do they need to be developed or can existing commercial solutions be purchased? If so, which ones? Maybe it would be more efficient to outsource the entire test effort to a specialized testing company and require only a skeleton testing crew to oversee their work.

Deciding on the strategy is a complex task—one that needs to be made by very experienced testers because it can determine the success or failure of the test effort. It's vitally important for everyone on the project team to understand and be in agreement with the proposed plan.

Test Schedule

The test schedule takes all the information presented so far and maps it into the overall project schedule. This stage is often critical in the test planning effort because a few highly desired features that were thought to be easy to design and code may turn out to be very time consuming to test. An example would be a program that does no printing except in one

limited, obscure area. No one may realize the testing impact that printing has, but keeping that feature

in the product could result in weeks of printer configuration testing time. Completing a test schedule as part of test planning will provide the product team and project manager with the information needed to better schedule the overall project. They may even decide, based on the testing schedule, to cut certain features from the product or postpone them to a later release.

Test Cases

The test planning process will decide what approach will be used to write them, where the test cases will be stored, and how they'll be used and maintained. Each test case must include the following phrases as per the ANSI/IEEE 829 standard

- **Identifiers.**
- **Test item.**
- **Input specification.**
- **Output specification.**
- **Environmental needs.**
- **Special procedural requirements.**
- **Inter case dependencies.**

4. a) b) What is meant by testing the pieces?

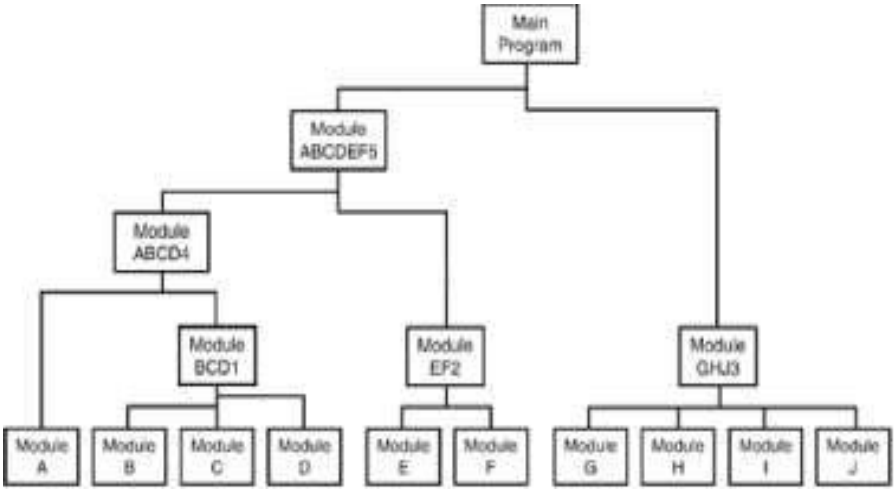
(Explanation of unit and integration testing 2 Marks each)

Ans: Testing the Pieces

This strategy covers two approaches: **unit testing and integration testing.**

Testing in pieces and gradually put together into larger and larger portions, there won't be any surprises when the entire product is linked together figure 1

Figure 1 Individual pieces of code are built up and tested separately, and then integrated and tested again.

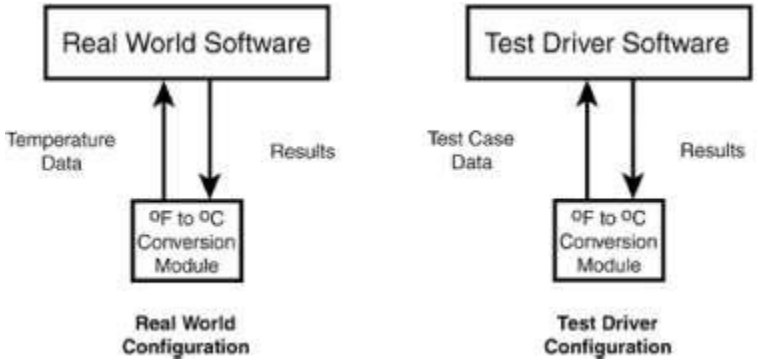


Testing that occurs at the lowest level is called unit testing or module testing. As the units are tested and the low-level bugs are found and fixed, they are integrated and integration testing is performed against groups of modules. **This process of incremental testing continues, putting together more and more pieces of the software until the entire product or at least a major portion of it is tested at once in a process called system testing.**

With this testing strategy, it's much easier to isolate bugs. When a problem is found at the unit level, the problem must be in that unit. If a bug is found when multiple units are integrated, it must be related to how the modules interact. Of course, there are exceptions to this, but by and large, testing and debugging is much more efficient than testing everything at once.

There are two approaches to this incremental testing: bottom-up and top-down. In bottom-up testing (see Figure 2), you write your own modules, called test drivers, that exercise the modules you're testing. They hook in exactly the same way that the future real modules will. These drivers send test-case data to the modules under test, read back the results, and verify that they're correct. You can very thoroughly test the software this way, feeding it all types and quantities of data, even ones that might be difficult to send if done at a higher level.

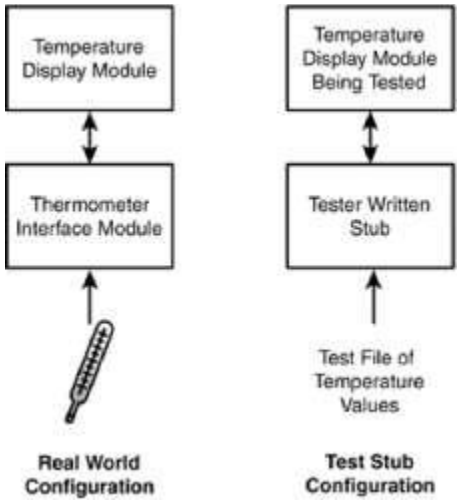
Figure 2. A test driver can replace the real software and more efficiently test a low-level module.



Top-down testing may sound like big-bang testing on a smaller scale. After all, if the higher-level software is complete, it must be too late to test the lower modules, right? Actually, that's not quite true.

In Figure 3, a low-level interface module is used to collect temperature data from an electronic thermometer. A display module sits right above the interface, reads the data from the interface, and displays it to the user. To test the top-level display module, you'd need blow torches, water, ice, and a deep freeze to change the temperature of the sensor and have that data passed up the line.

Figure 3. A test stub sends test data up to the module being tested.



Rather than test the temperature display module by attempting to control the temperature of the thermometer, you could write a small piece of code called a stub that acts just like the

interface module by feeding "fake" temperature values from a file directly to the display module. The display module would read the data and show the temperature just as though it was reading directly from a real thermometer interface module. It wouldn't know the difference. With this test stub configuration, you could quickly run through numerous test values and validate the operation of the display module.

4. a) c) Explain how to perform a website testing.

(Explanation – 4 Marks)

Ans: A partial list of web site testing includes

- **Text of different sizes, fonts, and colors (okay, you can't see the colors in this book)**
- **Graphics and photos**
- **Hyperlinked text and graphics**
- **Rotating advertisements**
- **Text that has Drop-down selection boxes**
- **Fields in which the users can enter data**

Figure 1. A typical web page has many testable features.

A great deal of functionality also isn't as obvious, features that make the website much more complex:

- **Customizable layout** that allows users to change where information is positioned onscreen
- **Customizable content** that allows users to select what news and information they want to see
- **Dynamic drop-down selection boxes**
- **Dynamically changing text**
- **Dynamic layout and optional information based on screen resolution**
- **Compatibility with different web browsers, browser versions, and hardware and software platforms**
- **Lots of hidden formatting, tagging, and embedded information that enhances the web** page's usability along with this the site is test with all the 3 types of testing techniques such as black box, white box and grey box testing .

Black box testing includes: text, hyperlinks, graphics, forms and objects.

White box testing includes Dynamic Contents, Database-Driven Web Pages, Programmatically Created Web Pages, Server Performance and Loading. Security etc parameter testing.

Grey box testing test the HTML code.

4. a) d) Describe about the localization issues.

(*Two points 2 Marks each with any appropriate example)*

Ans: Localization Issues

The process of adapting software to a specific locale taking into consideration its language, dialect, local conventions and culture is called localization.

Major elements for localization are:

1. Content

2. Data formats.

1. Content: If you are testing a product that will be localized, you need to carefully examine the content to make sure it‘s appropriate to the area where it will be used.

The following list shows various types of contents that you should carefully review for localization issues.

Sample document Icons

Pictures Sound

Video Help Files

Map Marketing materials

Packaging web links

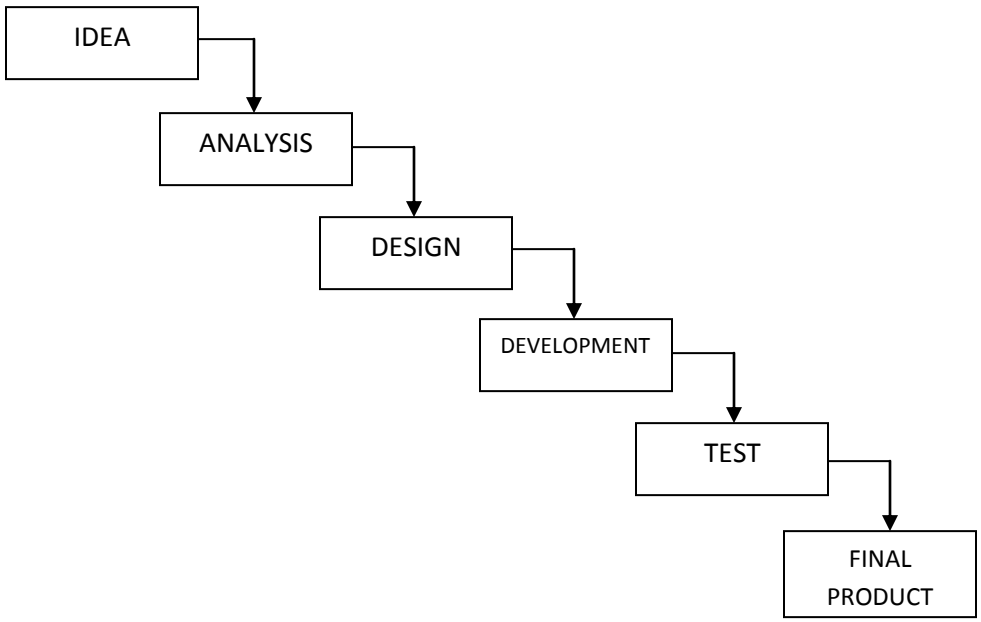
2. Data Formats: Different locals use different formats for data units such as currency, time & measurement.

4. b) Attempt any one of the following: Marks 6

a) Explain the Waterfall model with its diagram.

(Diagram - 2 Marks, Explanation- 4 Marks)

Ans: waterfall model:



Waterfall Model

This model is very simple & elegant. Fig. shows steps involved in this model.

A project using the waterfall model moves down a series of steps starting from an initial idea to final product. At the end of each step, the project team holds a review to determine if they are to move to the next step. It works well for projects with well-understood product definition.

Important thing about the waterfall method:

- There's large emphasis on specifying what product will be.
- The steps are discrete, there is no overlap.
- There is no way to go back up.

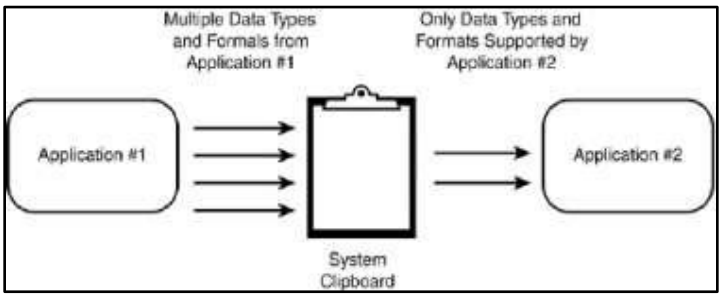
4. b) b) Explain Data sharing compatibility.

(Diagram – 2 Marks, Explanation of 4 points each 1 Mark)

Ans: **Data sharing compatibility is important to share the data among different applications.** A well written software product that supports and follows the published standards allows users to easily transfer data to and from other software product is a great compatible product.

The most common way of transferring the data from one program to other is saving and loading disk files. **The different examples of data sharing compatibility are:**

1. **File saves and File load is a common data sharing method.** You save data to a floppy disk and then carry it over to another computer running different software and load it in.
2. **Cut, Copy and paste is another method of sharing data among programs without transferring data to disk.** The transfer happens in memory through an intermediate program called the clipboard. Whenever the user performs the cut or copy the data that's chosen is placed in the clipboard. When he does a paste it is copied from the clipboard to the destination software. If you are compatibility testing program you need to make sure that the data is copied properly out of the clipboard to the desired program.



3. **DDE (Dynamic data exchange) ,COM(component object model) and OLE(object linking and embedding) are the common methods for transferring data between two applications.**

4. **File export and import:** It is the means that many programs use to be compatible with older versions of themselves and with other programs.

5. Attempt any two: Marks 16

a) Explain state testing with state transition diagram.

(Definition - 1 Mark, Diagram- 2 Marks, Explanation- 5 Marks)

Ans: The state testing is to verify program logic flow through its various states. A software state is condition or mode that software is currently in.

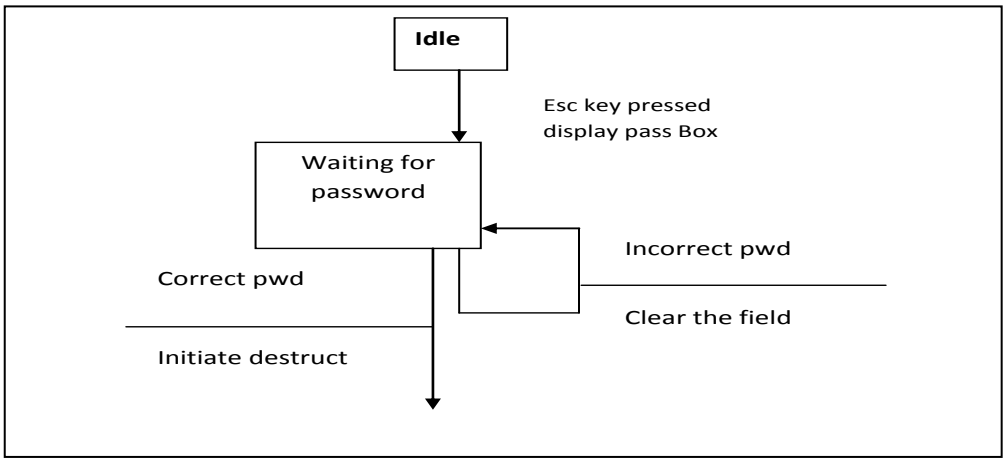
Testing the Software's Logic Flow

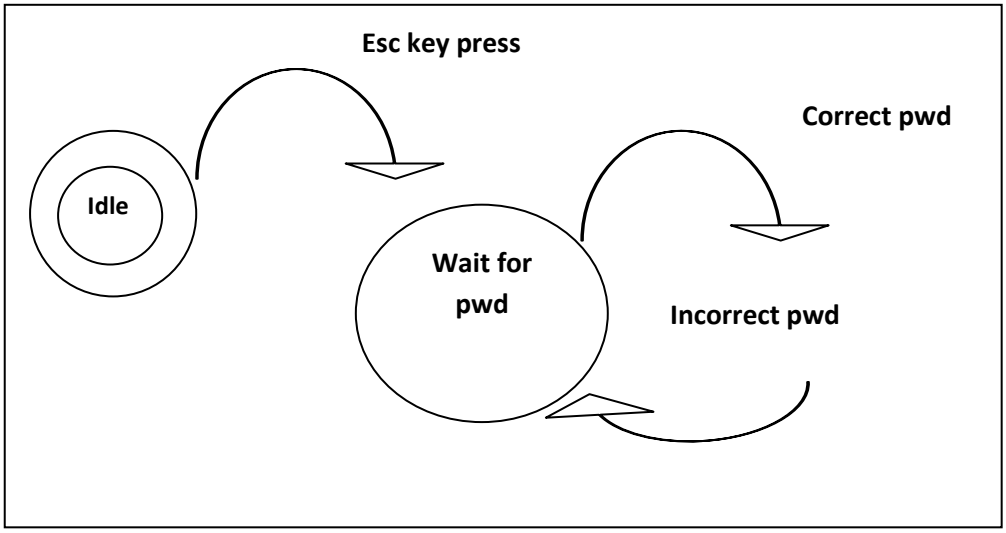
It's usually possible to visit all the states. The difficulty is that except for the simplest programs, it's often impossible to traverse all paths to all states To test hundreds or thousands of software states is very difficult.

The solution for software testing is to apply equivalence partition techniques to the selection of the states and paths, assuming some risk because you will choose not to test all of them, but reducing that risk by making intelligent choices.

Creating a State Transition Map

The first step is to create your own state transition map of the software. Such a map may be provided as part of the product specification. If you don't have a state map, you'll need to create one.





There are several different diagramming techniques for state transition diagrams. The figure3 above shows two examples. One uses boxes and arrows and the other uses circles (bubbles) and arrows. The technique you use to draw your map isn't important as long as you and the other members of your project team can read and understand it.

A state transition map should show the following items:

- Each unique state that the software can be in transition state.
- The input or condition that takes it from one state to the next. This might be a key press, a menu selection, a sensor input, a telephone ring, and so on. A state can't be exited without some reason. The specific reason is what you're looking for here.

- Set conditions and produced output when a state is entered or exited. This would include a menu and buttons being displayed, a flag being set, a printout occurring, a calculation being performed, and so on. It's anything and everything that happens on the transition from one state to the next.

Reducing the Number of States and Transitions to Test

- **Visit each state at least once.** It doesn't matter how you get there, but each state needs to be tested.
- **Test the state-to-state transitions that look like the most common or popular.** This sounds subjective, and it is, but it should be based on the knowledge you gained when you performed static black-box analysis of the product specification.
- **Test the least common paths between states.** It's likely that these paths were overlooked by the product designers and the programmers. You may be the first one to try them.
- **Test all the error states and returning from the error states.** Many times error conditions are difficult to create.

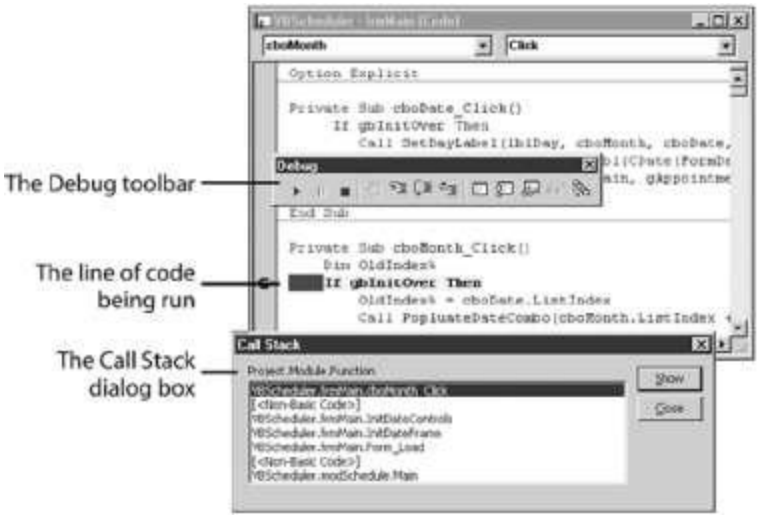
5. b) Explain the code coverage in detail.

(Definition 1 Mark, Explanation 4 Marks, Each coverage 1 Mark)

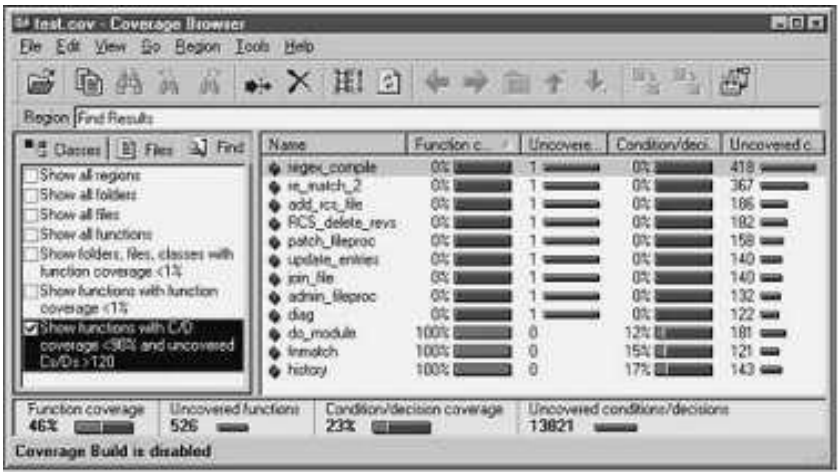
Ans: With testing the data, testing the program's states and the program's flow (coding) is also necessary. You must attempt to enter and exit every module, execute every line of code, and follow every logic and decision path through the software. This type of testing is known as code coverage testing.

Code coverage is dynamic white-box testing 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.

The simplest form of code coverage testing is using your compiler's debugger to view the lines of code you visit as you single-step through the program. Following figure shows an example of the Visual Basic debugger in operation.



For very small programs or individual modules, using a debugger is often sufficient. However, performing code coverage on most software requires a **specialized tool known as a code coverage analyzer as shown below.**



Code coverage analyzers hook into the software you're testing and run transparently in the background while you run your test cases. Each time a function, a line of code, or a logic decision is executed, the analyzer records the information. You can then obtain statistics that identify which portions of the software were executed and which portions weren't. With this data you'll know

- **What parts of the software your test cases don't cover**. If a specific module is never executed, you know that you need to write additional test cases for testing that module's function.
- **Which test cases are redundant?** If you run a series of test cases and they don't increase the percentage of code covered, they are likely in the same equivalence partition.
- **What new test cases need to be created for better coverage?** You can look at the code that has low coverage, see how it works and what it does, and creates new test cases that will exercise it more thoroughly.

Code coverage test points are:

Program statement and line coverage
Goal is to make sure that you execute every statement in the program at least once

```
1: PRINT "Hello World"
2: PRINT "The date is: "; Date$
3: PRINT "The time is: "; Time$
4: END
```

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

```
1: PRINT "Hello World"
2: IF Date$ = "01-01-2000" THEN
3:     PRINT "Happy New Year"
4:     END IF
5: PRINT "The date is: "; Date$
6: PRINT "The time is:"  Time$
7: END
```

Condition coverage:
It checks every condition from entire code. Condition coverage is shown in above example.

5. c) What is severity and priority of bugs? Explain a bugs life cycle in detail.

(Meaning- 2 Marks, Explanation with diagram- 6 Marks)

Ans: Severity: It indicates how bad the bug is, the likelihood and the degree of impact when the user encounters the bug. Levels are

1. System crash, data loss, data corruption
2. Operational error, wrong result, loss of functionality
3. Minor problem, misspelling, UI layout, rare occurrence

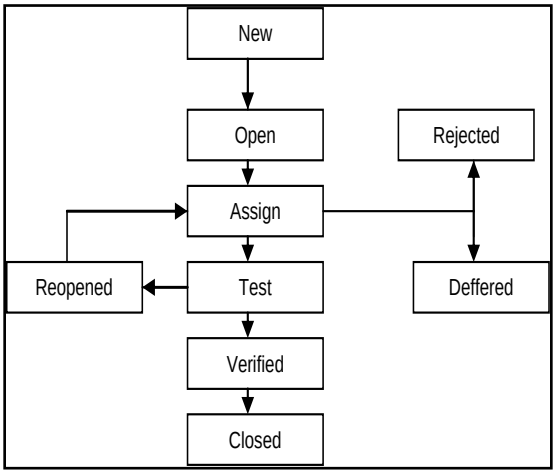
Priority: It indicates how much emphasis should be placed on fixing the bug and the urgency of making the fix. Levels are

1. Immediate fix, blocks further testing, very visible
2. Must fix before the product is released
3. Should fix when time permits
4. Would like to fix but the product can be released as it is

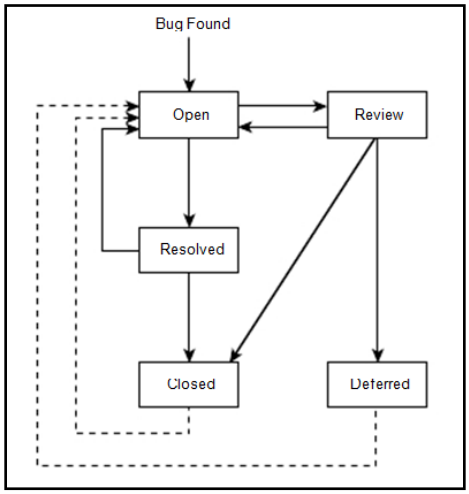
Explain the bug life cycle

Software problems are called bugs, when tester founds the bug, that bug is reported or entered bug into database. In software development the bug has a life cycle.

Following figure shows the bug life cycle:



OR



The different states of bug life cycle are as shown in the above diagram:

1. New: When the bug is posted for the first time, its state will be "NEW". This means that the bug is not yet approved.

2. Open: After a tester has posted a bug, the lead of the tester approves that the bug is genuine and he changes the state as "OPEN".

3. Assign: Once the lead changes the state as "OPEN", he assigns the bug to corresponding developer or developer team. The state of the bug now is changed to "ASSIGN".

4. Test: Once the developer fixes the bug, he has to assign the bug to the testing team for next round of testing. Before he releases the software with bug fixed, he changes the state of bug to "TEST". It specifies that the bug has been fixed and is released to testing team.

5. Deferred: The bug, changed to deferred state means the bug is expected to be fixed in next releases. The reasons for changing the bug to this state have many factors. Some of them are priority of the bug may be low, lack of time for the release or the bug may not have major effect on the software.

6. Rejected: If the developer feels that the bug is not genuine, he rejects the bug. Then the state of the bug is changed to "REJECTED".

7. Verified: Once the bug is fixed and the status is changed to "TEST", the tester tests the bug. If the bug is not present in the software, he approves that the bug is fixed and changes the status to "VERIFIED".

8. Reopened: If the bug still exists even after the bug is fixed by the developer, the tester changes the status to "REOPENED". The bug traverses the life cycle once again.

9. Closed: Once the bug is fixed, it is tested by the tester. If the tester feels that the bug no longer exists in the software, he changes the status of the bug to "CLOSED". This state means that the bug is fixed, tested and approved.

6. Attempt any four of the following : Marks 16

a) How to identify hardware standards?

(Each points - 1 Mark)

Ans:

- Reviewing the specification that the hardware companies use to create their products-you can look in a couple of places.
- Knowing some details of the hardware specifications can help you make more informed equivalence partition decision.
- Visit hardware website about developing and testing hardware in device driver
- Regarding configuration testing to indentify hardware, same questions asked.
  - What external hardware will operate with this software?
  - What models and versions of that hardware are available?

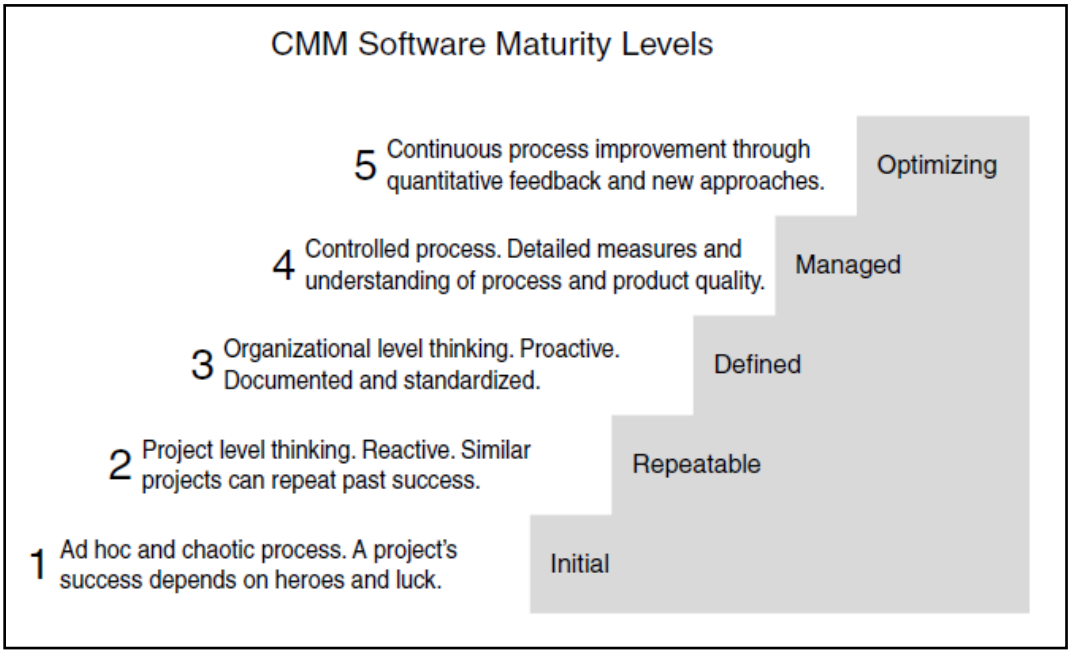
- What features or option does that hardware support?

6. b) Explain Capability Maturity model (CMM).

(Diagram-1 Mark, Explanation-3 Marks)

Ans: Capability Maturity Model (CMM)

The Capability Maturity Model for Software (CMM) is an industry-standard model for defining and measuring the maturity of a software company's development process and for providing direction on what they can do to improve their software quality. It provides a means to assess a company's software development maturity and determine the key practices they could adopt to move up to the next level of maturity.



• **Level 1**: **Initial.** The software development processes at this level are ad hoc and often chaotic. The project's success depends on heroes and luck. There are no general practices for planning, monitoring, or controlling the process. It's impossible to predict the time and cost to develop the software. The test process is just as ad hoc as the rest of the process.

• **Level 2**: **Repeatable.** This maturity level is best described as project-level thinking.

Basic project management processes are in place to track the cost, schedule, functionality, and quality of the project. Lessons learned from previous similar projects are applied. There's a sense of discipline. Basic software testing practices, such as test plans and test cases, are used.

• **Level 3: Defined.** Organizational, not just project specific, thinking comes into play at this level. Common management and engineering activities are standardized and documented. These standards are adapted and approved for use on different projects. The rules aren't thrown out when things get stressful. Test documents and plans are reviewed and approved before testing begins. The test group is independent from the developers. The test results are used to determine when the software is ready.

• **Level 4: Managed**. At this maturity level, the organization's process is under statistical control. Product quality is specified quantitatively beforehand (for example, this product won't release until it has fewer than 0.5 defects per 1,000 lines of code) and the software isn't released until that goal is met. Details of the development process and the software's quality are collected over the

project's development, and adjustments are made to correct deviations and to keep the project on plan.

• **Level 5: Optimizing.** This level is called Optimizing (not "optimized") because it's continually improving from Level 4. New technologies and processes are attempted, the results are measured, and both incremental and revolutionary changes are instituted to achieve even better quality levels. Just when everyone thinks the best has been obtained, the crank is turned one more time, and the next level of improvement is obtained.

6. c) What is formal review?

(Each point 1 Mark)

Ans:

- 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 design or code, not the person who created it.
- 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.
- 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 fulfill them at the review. Most of the problems found through the review process are found during preparation, not at the actual review.
- 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.

6. d) Explain how to test for disabled.

(Each point – 1 Mark)

Ans: A topic that falls under the area of usability testing is that of accessibility testing or testing for disabled.

There are many types of disabilities, the following ones make using computers and software especially difficult:

Visual impairments: Color blindness, extreme near and far sightedness, tunnel vision, dim vision, blurry vision, and cataracts are examples of visual limitations. People with one or more of these would have their own unique difficulty in using software. Think about trying to see

where the mouse pointer is located or where text or small graphics appear onscreen. What if you couldn't see the screen at all?

Hearing impairments: Someone may be partially or completely deaf, have problems hearing certain frequencies, or picking a specific sound out of background noise. Such a person may not be able to hear the sounds or voices that accompany an onscreen video, audible help, or system alerts.

Motion impairments: Disease or injury can cause a person to lose fine, gross, or total motor control of his hands or arms. It may be difficult or impossible for some people to properly use a keyboard or a mouse. For example, they may not be able to press more than one key at a time or may find it impossible to press a key only once. Accurately moving a mouse may not be possible.

Cognitive and language: Dyslexia and memory problems may make it difficult for someone to use complex user interfaces. Think of the issues outlined previously in this chapter and how they might impact a person with cognitive and language difficulties.

Due to all above reasons to access software with help of hardware's by disabled user accessibility testing is needed.

6. e) Explain sub-boundary conditions in data testing.

(Explanation 2 Marks, Any one Example 2 Marks)

Ans: Sub-Boundary Conditions: Some boundaries, though, that are internal to the software aren't necessarily apparent to an end user but still need to be checked by the software tester. These are known as sub-boundary conditions or internal boundary conditions. Two examples are Powers-Of-Two and the ASCII table.

Powers-of-Two:

Computers and software are based on binary numbers-bits representing 0s and 1s, bytes (8 bits), words (on 32-bit systems) made up of 4 bytes, and so on. The following table shows the common powers-of-two units and their equivalent values.

Term	**Range or Value**
Bit	0 or 1
Nibble	0-15
Byte	0-255
Word	0-4,294,967,295
Kilo	1,024
Mega	1,048,576
Giga	1,073,741,824
Tera	1,099,511,627,776

ASCII Table

Another common sub-boundary condition is the ASCII character table. Following table is a partial listing of the ASCII table.

Character	ASCII Value	Character	ASCII Value
Null	0	B	66
Space	32	Y	89
/	47	Z	90
0	48	[	91
1	49	'	96
2	50	a	97
9	57	b	98
:	58	y	121
@	64	z	122
A	65	{	123

0 through 9 are assigned to ASCII values 48 through 57. The slash character, /, falls before 0. The colon, :, comes after 9. The uppercase letters A through Z go from 65 to 90. The lowercase letters span 97 to 122. All these cases represent sub-boundary conditions.