

I. Examining the Specification

The Software Development Process: big-bang, code-and-fix, waterfall, and spiral. In each model, except big-bang, the development team creates a product specification, sometimes called a requirements document, to define what the software will become.

1. Black-Box and White-Box Testing (Question: Explain black box testing and white box testing – 2 marks each = 4 marks)

Black Box Testing

1. This approach tests all possible combinations of end-user actions.
2. Black box testing assumes no knowledge of code and is intended to simulate the end-user experience.
3. The tester can use sample applications to integrate and test the application block for black box testing.
4. Planning for black box testing immediately after the requirements and the functional specifications are available.
5. Black box testing should be conducted in a test environment close to the target environment.

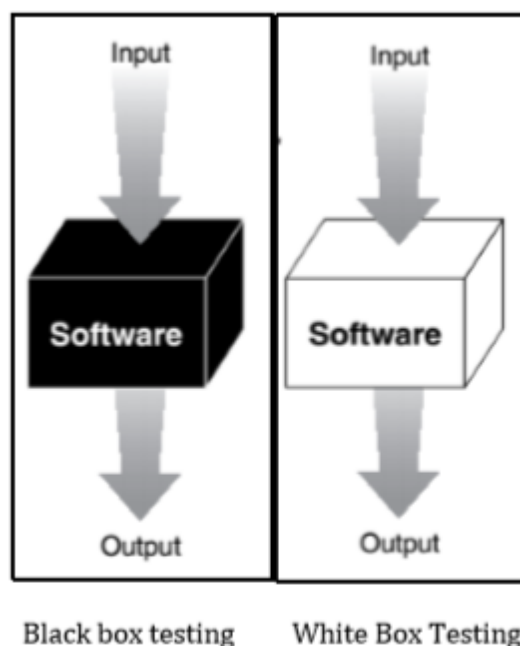


Figure 2.1

White box testing

1. This is also known as glass box, clear box, and open box testing.
2. In white box testing, test cases are created by looking at the code to detect any potential failure scenarios.
3. The suitable input data for testing various APIs and the special code paths that need to be tested by analysing the source code for the application block.
4. Therefore, the test plans need to be updated before starting white box testing and only after a stable build of the code is available.
5. White box testing assumes that the tester can take a look at the code for the application block and create test cases that look for any potential failure scenarios.
6. During white box testing, analyze the code of the application block and prepare test cases for testing the functionality to ensure that the class is behaving in accordance with the specifications and testing for robustness.
7. A failure of a white box test may result in a change that requires all black box testing to be repeated and white box testing paths to be reviewed and possibly changed.

2. **Static and Dynamic testing** (Question: Explain static testing and dynamic testing = 4 marks)

1. Dynamic testing is testing that is performed when the system is running.
2. The basic requirement is to review test plans.
3. Recommend tests based on the hazard analyses, safety standards and checklists, previous accident and incidents, operator task analyses etc.
4. Specify the conditions under which the test will be conducted.
5. Review the test results for any safety-related problems that were missed in the analysis or in any other testing.
6. Ensure that the testing feedback is integrated into the safety reviews and analyses that will be used in design modifications
7. Static testing is performed when the system is not running.
8. Static testing works with peer review, and mostly referred to as pen and pencil run.

3. Static black box testing: - Testing the specification. (Question: Explain testing specification in static testing = 4 marks)

1. Testing the specification is static black-box testing.
2. The specification is a document, not an executing program, so it's considered static.
3. It's also something that was created using data from many sources—usability studies, focus groups, marketing input, and so on.
4. It is not necessarily to know how or why that information was obtained or the details of the process used to obtain it, just that it's been boiled down into a product specification.
5. Tester can then take that document, perform static black-box testing, and carefully examine it for bugs.

a) **Performing a High-Level Review of the Specification** (Question: Explain performance a high level review of specification static testing = 8 marks)

1. The first step is to stand back and view it from a high level. Examine the specifications for large fundamental problems, oversights, and omissions.
2. It is important to do research before testing, but ultimately the research is a means to better understand what the software should do.
3. Research leads to the better understanding of the product on which testing is to be performed.
4. For performing the high level review the guidelines are for pretending to be a customer, research existing guidelines and review the similar test cases.

- Pretend to Be the Customer

1. The easiest thing for a tester to do when he first receives a specification for review is to pretend to be the customer.
2. Learn about the area where it will be used and implemented it's important to understand the customer's expectations.

3. As the quality means “meeting the customer’s needs.” As a tester, to understand those needs and to test the software is important.

4. The tester should be an expert in the field of testing.

- **Research Existing Standards and Guidelines**

1. Research to be done on the existing similar type of software’s to gain Knowledge about the various factors that affect its working.

2. Effort to standardize the hardware and the software is important.

3. The result is that we now have products reasonably similar in their look and feel that have been designed.

4. The guidelines are followed for the existing software’s are implemented.

- **Review and Test Similar Software Some things to look for when reviewing competitive products include**

1. Scale: - Will your software be smaller or larger? Will that size make a difference in your testing? As the size of the software increases the hardware requirement also changes. Keeping in mind the system requirement the software has to be developed.

2. Complexity: - Will your software be more or less complex? Will this impact your testing? The data structure used has to be pre-decided to reduce the complexity in the program and modules should be well defined.

3. Testability: -Will you have the resources, time, and expertise to test software such as this? The software has to be tested unit by unit and then the complete integration testing to be performed for the product to be ready for deployment.

4. Quality/Reliability: -Is this software representative of the overall quality planned for your software? Will your software be more or less reliable? The update of the product to be made available and the proper documentation to be prepared for future reference.

b) Low-Level Specification Test Techniques (Question: Explain low level specification in static testing = 8 marks)

After you complete the high-level review of the product specification the testing of the low level specification is done.

- **Specification Attributes Checklist** A good, well-thought-out product specification has eight important attributes:

1. Complete: -Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
2. Accurate: -Is the proposed solution correct? Does it properly define the goal? Are there any errors?
3. Precise, Unambiguous, and Clear: -Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understandable?
4. Consistent: -Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
5. 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?
6. Feasible: -Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
7. Code-free: -.Does the specification stick with defining the product and not the underlying software design, architecture, and code?
8. 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. The appearance of these words often signifies that a feature isn't yet

1. Completely thought out: - it likely falls under one of the preceding attributes. Look for these words in the specification and carefully review how they're used in context. The spec may go on to clarify or elaborate on them, or it may leave them ambiguous—in which case, you've found a bug.

2. 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.

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

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

5. 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.

6. 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.

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

8. 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.

II. Testing the Software with Blinders On

1. It is very easy to start testing the software without knowing and just rectify the errors as and when encountered.

2. Such an approach might work for a little while. If the software is still under development, it's very easy to get lucky and find a few bugs right away.

3. Unfortunately, those easy pickings will quickly disappear and a more structured and targeted approach is required to continue finding bugs and to be a successful software tester.

4. The testing approach has to be designed for it to work perfectly and for a successful software tester.

1. Dynamic Black-Box Testing a) Testing the Software while Blindfolded (Question: Explain the concept of testing the software blindfolded in dynamic black box testing = 4 marks)

1. Testing software without having an insight into the details of underlying code is dynamic black-box testing. 2. It's dynamic because the program is running and testing is done simultaneously using it as a customer would. And, it's black-box because you're testing it without knowing exactly how it works—with blinders on. 3. The inputs are entered, and receiving outputs and checking the results are tabulated. 4. Another name commonly used for dynamic black-box testing is behavioral testing because you're testing how the software actually behaves when it's used.

b) **Test-to-Pass and Test-to-Fail** (Question: Explain the concept of test to pass and

test to fail in dynamic black box testing = 4 marks)

There are two fundamental approaches to testing software test-to-pass and test-to-fail.

1. When you test-to-pass, you really assure only that the software minimally works and as a software tester it is the first positive approach.
2. Don't push its capabilities; don't see what you can do to break it. Test the software by applying the simplest and most straightforward test cases.
3. The software should be tested for it to work under the best ideal condition.
4. During test to fail, the software should be tested for the boundary conditions on its various limits. 5. If the program is using integer numbers, then the output should be tested in the limits of the integer variable.

For example

```
1. #include<stdio.h>
2. void main()
3. {
4. int i , fact= 1, n;
5. printf("enter the number ");
6. scanf("%d",&n);
7. for(i =1 ;i <=n;i++)
8. fact = fact * i;
9. printf ("the factorial of a number is %d", fact);
10. }
```

The above program works for the integer values from 1 to 7, and fails as soon as the input value is entered as 8, as it exceeds the integer range of variable for fact.

c) Equivalence Partitioning

1. Type of black box testing that divides that divides the input domain of the program into classes of data from which the test cases can be divided.
2. This method is typically used to reduce the total number of test cases to a finite set of testable test cases, still covering maximum requirements.

3. It is the process of taking all possible test cases and placing them into classes. One test value is picked from each class while testing.

For example: If you are testing for an input box accepting numbers from 1 to 1000 then there is no use in writing thousand test cases for all 1000 valid input numbers plus other test cases for invalid data.

1. Using equivalence partitioning method above test cases can be divided into three sets of input data called as classes. Each test case is a representative of respective class.

2. For the above example we can divide our test cases into three equivalence classes of some valid and invalid inputs.

- Test cases for input box accepting numbers between 1 and 1000 using Equivalence Partitioning:

1. One input data class with all valid inputs. Pick a single value from range 1 to 1000 as a valid test case. If you select other values between 1 and 1000 then result is going to be same. So one test case for valid input data should be sufficient.

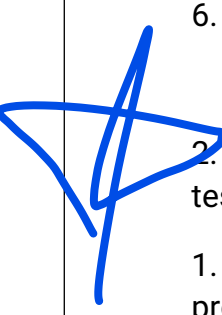
2. Input data class with all values below lower limit. i.e. any value below 1, as an invalid input data test case.

3. Input data with any value greater than 1000 to represent third invalid input class.

4. So using equivalence partitioning all possible test cases is partitioned into three classes. Test cases with other values from any class should give you the same result.

5. We have selected one representative from every input class to design our test cases. Test case values are selected in such a way that largest number of attributes of equivalence class can be exercised.

6. Equivalence partitioning uses fewest test cases to cover maximum requirements.

 2. **Data Testing** (Question: Explain the concept data testing in dynamic black box testing = 4 marks)

1. The simplest view of software is to divide its world into two parts: the data and the program.

2. The data is the keyboard input, mouse clicks, disk files, printouts, and so on.

3. The program is the executable flow, transitions, logic, and computations.

4. When software testing is performed on the data, the user information is checked and the data is tabulated with the expected results.

Examples of data would be

- The words you type into a word processor
- The numbers entered into a spreadsheet
- The number of shots you have remaining in your space game
- The picture printed by your photo software
- The backup files stored on your floppy disk
- The data being sent by your modem over the phone lines

a) **Boundary Condition** (Question: Explain the concept of boundary conditions and sub boundary conditions in dynamic black box testing – 2 marks each = 4 marks)

1. Boundary conditions are special because programming, by its nature, is susceptible to problems at its edges.
2. The boundary conditions are defined as the initial and the final data ranges of the variables declared.
3. If an operation is performed on a range of numbers, odds are the programmer got it right for the vast majority of the numbers in the middle, but maybe made a mistake at the edges.
4. The edges are the minimum and the maximum values for that identifier.

For example

1. `#include<stdio.h>`
2. `void main()`
3. `{`
4. `int i , fact=1, n;`
5. `printf("enter the number ");`
6. `scanf("%d",&n);`
7. `for(i =1 ;i <=n;i++)`

```
8. fact = fact * i;  
9. printf ("the factorial of a number is 是 "%d", fact);  
10. }
```

The boundary condition in the above example is for the integer variable.

b) Sub-Boundary Conditions

1. They're the ones defined in the specification or evident when using the software.
2. 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.
3. These are known as sub-boundary conditions or internal boundary conditions.
4. In the given example the sub boundary condition is the value of factorial

For example

```
1. #include<stdio.h>  
2. void main()  
3. {  
4. int i , fact=1, n;  
5. printf("enter the number ");  
6. scanf("%d",&n);  
7. for(i =1 ;i <=n;i++)  
8. fact = fact * i;  
9. printf ("the factorial of a number is 是 "%d", fact);  
10. }
```

c) Default, Empty, Blank, NULL, Zero and None, Invalid, Wrong, Incorrect and Garbage Data

1. The final type of data testing is garbage data. This is where you test-to-fail.
2. You've already proven that the software works as it should by testing-to-pass with boundary testing, sub boundary testing, and default testing.
3. Now it's time to throw the trash at it.

4. Software testing purists might argue that this isn't necessary, that if you've tested everything discussed so far you've proven the software will work.

5. In the real world, however, there's nothing wrong with seeing if the software will handle whatever a user can do to it.

3. **State Testing** (Question: Explain the concept state testing in dynamic black box testing –4 marks)

1. The data gets tested on the numbers, words, inputs, and outputs of the software.
2. The product or the software should be tested for the program's logic flow through its various states.
3. A software state is a condition or mode that the software is currently in.
4. Consider Figures 2.2, which illustrates the software mode of the paint software in airbrush mode.

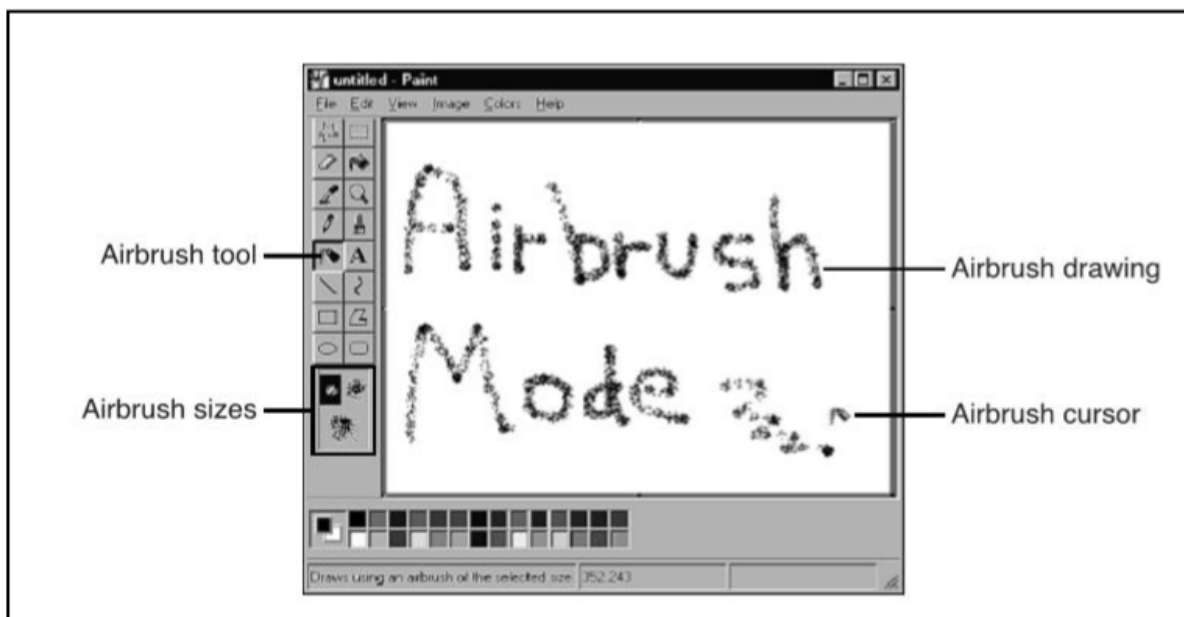


Figure 2.2 a)

Testing Software's Logic Flow

1. Testing the software's states and logic flow has the same problems. It's usually possible to visit
2. Check for the all possible states for test to pass condition
3. The complexity of the software, especially due to the richness of today's user interfaces, provides so many choices and options that the number of paths grows

exponentially.

- **Creating a State Transition Map** (Question: Explain the concept of creating a state transition map. Diagram – 1 mark, Explanation 3 marks = 4 marks)

1. There are various techniques for state transition diagrams.
2. Figure 2.3 shows two examples. One uses boxes and arrows and the other uses circles (bubbles) and arrows.
3. The state transition diagram should be easily understandable and should be able to explain clearly about the various stages the software passes through.

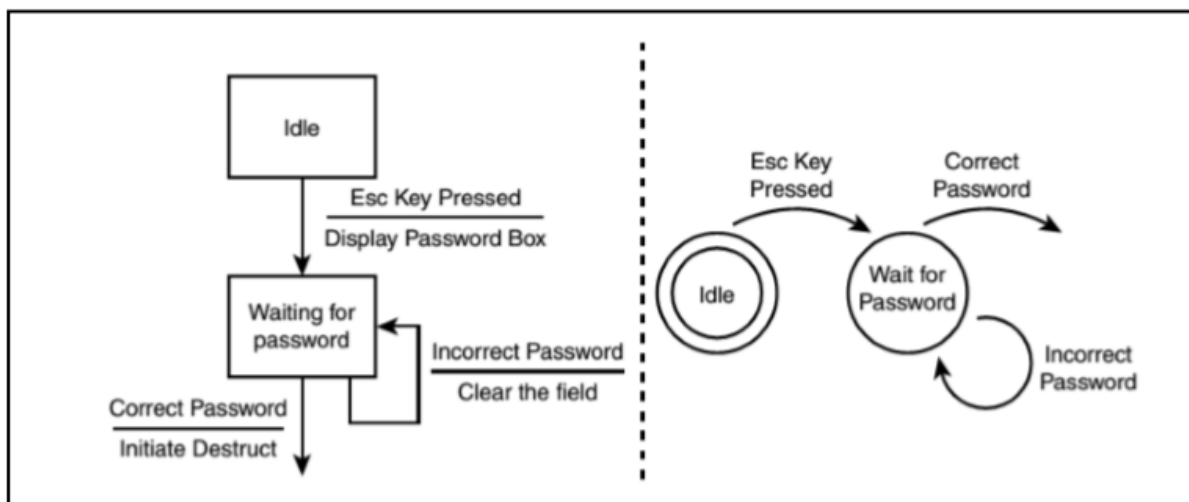


Figure 2.3

A state transition map should show the following items:

1. Each unique state that the software can be in.
2. A good rule of thumb is that if the state is unsure whether something is a separate state, it probably is.
3. Always collapse it into another state if you find out later that it isn't.
2. The input or condition that takes it from one state to the next.
3. This might be a key press, a menu selection, a sensor input, a telephone ring, and so on.
4. A state can't be exited without some reason. The specific reason is what you're looking for here.
5. Set conditions and produced output when a state is entered or

exited.

6. This would include a menu and buttons being displayed, a flag being set, a printout occurring, a calculation being performed, and so on.

7. It's anything and everything that happens on the transition from one state to the next.

b) **Testing States** to Fail Testers while performing takes care about the various aspects of operating system and the various testing methods like stress test, load test and regression testing.

- **Race Conditions and Bad Timing** (Question: Explain what leads to race condition in software = 4 marks)

1. As operating systems can do multitasking.

2. Multi-tasking means that an operating system is designed to run separate processes concurrently. 3. These processes can be separate programs such as a spreadsheet

For example

1. Saving and loading the same document at the same time with two different programs.

2. Sharing the same printer, communications port, or other peripheral

3. Pressing keys or sending mouse clicks while the software is loading or changing states

4. Shutting down or starting up two or more instances of the software at the same time

5. Using different programs to simultaneously access a common database.

- **Repetition, Stress, and Load**

1. The software fails under are repetition, stress, and load.

2. These tests target state handling problems where the programmer didn't consider what might happen in the worst-case scenarios.
3. Repetition testing involves doing the same operation over and over.
4. This could be as simple as starting up and shutting down the program over and over.
5. It could also mean repeatedly saving and loading data or repeatedly selecting the same operation.
6. A bug can be encountered after only a couple repetitions or it might take thousands of attempts to reveal a problem.
7. The repetition testing is done to look for memory leaks.
8. A common software problem happens when computer memory is allocated to perform a certain operation but isn't completely freed when the operation completes.
9. The result is that eventually the program uses up memory that it depends on to work reliably.
10. Stress testing is running the software under less-than-ideal conditions such as low memory, low disk space, slow CPUs, slow modems.
11. The software should be tested for external resources and dependencies it has on them.
12. Stress testing is simply limiting them to their bare minimum.

- **Other Black-Box Test Techniques**

These techniques are the combinations of the various tests and methods and the specification details that are been listed before in the chapter.

1. Behave like a dumb user: - As a tester behave like you are new to the field and don not know anything to do with the product or the software that you are testing.
2. Look for bugs where you have already found them: - As a tester if you have already worked on the similar project then just jump to the places where you know that the bugs can be found. Bugs are likely to occur at the same place.
3. Follow Experience, Intuition, and Hunches: - the experience of the test gives the tester an advantage to find the bugs early and also how to remove the bugs as early as possible.