Principle of testing

There are seven principles of testing

1. Testing shows the presence of Defects
2. Exhaustive testing is not possible
3. Early testing
4. Defect clustering
5. Pesticide paradox
6. Testing is context-dependent
7. Absence-of-errors fallacy

1.Testing shows the presence of defects

You test software to identify problems so you can fix them before you deploy the software to production environments. However, this process doesn't mean that there aren't any bugs in the product. It just means that there may be bugs, but you didn't find them.

There could be any number of reasons that you didn't uncover every bug, including the fact that the test cases didn't cover every scenario.

This principle, which helps to set stakeholder expectations, means that you shouldn't guarantee that the software is error-free.

2.Exhaustive testing is impossible

The truth is that you can't test everything, i.e., every combination of preconditions and inputs. And if you try to do so you'll waste time and money, but it won't affect the overall quality of the software.

What you need to do is assess risk and plan your tests around these risks so you can be sure you're testing the key functions. Careful planning and assessment ensures your test coverage is good so you can have confidence in your final product — and you don't even have to test every individual line of code.

3.Early testing

When it comes to the software development lifecycle, testing early is the key to identifying any defects in the requirements or design phase as soon as possible. It's much easier and less expensive to fix bugs in the early stages of testing than at the end of the software lifecycle as then you might have to rewrite entire areas of functionality. And that likely means missed deadlines and cost overruns.

4.Defect clustering

Defect clustering is the idea that a small number of software modules or components contain the most defects — sort of applying the Pareto Principle to software testing, i.e., approximately 80% of the issues are found in 20% of the components.

Understanding this can help in your testing because if you find one defect in a particular area, you'll likely find more in that same module. If you identify the complex areas that are changing the most or the ones that have more dependencies, you can focus your testing on these key areas of risk.

5.Pesticide paradox

This principle centers around the theory that if you repeatedly use a particular pesticide on your crops, the insects you're trying to kill or repel will eventually become immune to the pesticide and it will no longer be effective.

Likewise, if you continuously run the same tests, eventually they'll fail to find new defects, even though they'll probably confirm the software is working.

Consequently, you must continue to review your tests as well as add to your scenarios or modify them to help prevent this pesticide paradox. For example, maybe you could use a variety of testing techniques, methods, and approaches simultaneously.

6.Testing is context dependent

Software testing is all about the context, which means that no one strategy will fit every scenario. The types of testing and the methods you use totally depend on the context of the systems or the software, e.g., the testing of an iOS application is different from the testing of an e-commerce website. Put simply, what you're testing will always affect the approach you use.

7.Absence-of-errors fallacy

If your software is 99% error-free but it doesn't follow your user's requirements, it's still not usable. That's why it's critical to run tests that pertain to the requirements of the system. Software testing isn't just about finding bugs, it's about ensuring that the software meets the user's needs and requirements.

As such, you should also test your software with the users. You can test against early prototypes at the usability testing phase so you can get feedback from the users that you can use to ensure the software is usable. Even though your software might have relatively few issues, doesn't mean it is ready to ship; it also has to meet your customer's requirements and expectations.

#Black Box Testing and White Box Testing

Black box Testing

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioural Testing.

Black Box Testing Techniques

Following the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

Types of Black Box Testing

There are many types of Black Box Testing but the following are the prominent ones –

- **Functional testing** – This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** – This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** –Regression testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

White Box Testing

White Box Testing is a testing technique in which software's internal structure, design, and coding are tested to verify input-output flow and improve design, usability, and security. In white box testing, code is visible to testers, so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing, and Glass box testing.

Important White Box Testing Techniques:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage

- Multiple Condition Coverage
- Finite State Machine Coverage
- Path Coverage
- Control flow testing
- Data flow testing

Statement Coverage

Statement coverage is a white box testing technique that ensures all executable statements in the code are run and tested at least once. For example, if there are several conditions in a block of code, each of which is used for a certain range of inputs, the test should execute each and every range of inputs, to ensure all lines of code are actually executed.

Branch Coverage

Branch coverage maps the code into branches of conditional logic, and ensures that each and every branch is covered by unit tests.

Path Coverage

Path coverage is concerned with linearly independent paths through the code.

Code coverage

Code coverage is a metric that shows how much of an application's code has unit tests checking its functionality.

Types of White Box Testing

White box testing encompasses several testing types used to evaluate the usability of an application, block of code or specific software package. There are listed below —

- **Unit Testing:** It is often the first type of testing done on an application. Unit Testing is performed on each unit or block of code as it is developed. Unit Testing is essentially done by the programmer. As a software developer, you develop a few lines of code, a single function or an object and test it to make sure it works before continuing Unit Testing helps identify a majority of bugs, early in the software development lifecycle. Bugs identified in this stage are cheaper and easy to fix.

- **Testing for Memory Leaks**: Memory leaks are leading causes of slower running applications. A QA specialist who is experienced at detecting memory leaks is essential in cases where you have a slow running software application.