**Test coverage and its importance**

**What is test coverage?**

In the process of software testing, our job is not just limited to test the software but also to have a track on to what extent the validation is done. Basically the test coverage defines how much functionality of the application has been tested? This metric is basically be calculated as the number of test case executed to the total number of tests identified.

**Importance of test coverage**

More the coverage more it emphasizes that the thorough testing is done. Given the scope and process followed in each type of testing we have coverage calculated in its own fashion

The coverage check should be considered from the initial level, for instance checking for the code coverage during unit testing will help us understanding the all the code written was reached and that way we can identify if there is a redundant code then remove accordingly

It helps developers to understand whether the decision flow that’s written has been executed at least once in one or other scenario.

During testing phase upon considering the proper techniques, we can cover the extremes of the application with as many less number of test cases as possible

Let’s cut to the chase, below are some of the types of test coverage we have in different types of testing.

**White box testing techniques**

White box testing a test designed or executed on the structure or design of an application and proper understanding of the code and system is must.

In this type of testing the coverage techniques used are,

**Statement coverage**

Let’s consider a developer is conducting a unit testing, developer must have thorough understanding of what that module or component should do? Also developer must aware of the code that has been written. Predominantly with the help of this technique we try to get rid of any unreachable code that’s available in the system. Ideally the statement coverage is equal to the number of statements being written.

Eg:

Main(){

If(a>0&&b>0){

If(a>b){

Print is bigger;

Else

Print b is bigger;

}

Else{

Print error message saying, negative values are not allowed;

}

}

The statement coverage for the above program is 9, as there are 9 statements

**Decision coverage**

The decision coverage mechanism of a white box testing technique ensures that we make proper conditional calls and don’t miss any edge cases. The decision coverage is equal to the number of decisions we have in the system

Eg:

Main(){

If(a>0&&b>0){

If(a>b){

Print is bigger;

Else

Print b is bigger;

}

Else{

Print error message saying, negative values are not allowed;

}

}

In the above snippet we are making two decisions as whether or not both the given numbers are non-negative integers then considering a decision as which number is bigger

So the decision coverage for the snippet is equals to 2

**Path coverage**

Path coverage ensure that how many use cases and flows can be derived based on the given inputs. Also we need to ensure that all possible paths are covered to ensure that system is not open to break in any scenario. This can be understood by drawing some UML, flow chart diagrams.

**Black box testing techniques**

Black box is a manual validation and test cases are mainly focused to test the functional validation. These techniques will help us to do some permutations and combinations and identify the test cases based on the edge cases or doing some partitioning

**Boundary value analysis**

In this technique based on the condition, the boundary values such one immediate left and one immediate right boundary values are tested to ensure the state transition is happening properly on the conditional edge.

Eg lets consider that a person whose age is 18 or above only can vote and below age group people can’t cast their vote.

In order to test this condition we will understand the deciding factor age as “18” as its mentioned then identify one value just below 17 and one above 19 as boundary values to test that critical conditional logic

**Equivalence partitioning**

This is another technique, which will help us to avoid the number of test cases but gives more coverage in this testing. In this technique based on the condition we will partition the negative and positive slots and then pick one of those values to test. The lore is that, if the test case is behaving as expected for one value it is understood rest other values would be passed from the same slot.

Eg: let’s consider the same example as above that person with 18 or above can vote and rest can’t

We will identify three partitions as

Age- 0<=17- negative slot

Age- 18= positive slot

Age- 19 to any greater value- positive slot

We will pick any one test data from each classification then with the help of 3 test cases all the scenarios can be covered.

**Decision tables**

When we have multiple combinations of inputs to test, if we choose to write test cases for all the values it becomes an overhead. Then forming a table with all the conditions and actions put in would help better to identify the cases.

Eg: let’s consider a scenario of withdrawing money from the automated teller machine.

The machine will dispense money only when

Cond-1 user put in correct passcode

Cond-2 withdrawal money should be lesser or equal to the savings balance.

**Action / Goal :** withdraw money from the teller machine.

The above requirement can be converted into a decision table as,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Condition** | **Comb-1** | **comb-2** | **comb-3** | **comb-4** |
| pass code should be correct | T | F | T | F |
| Withdrawal amount<=savings | T | T | F | F |
| **Actions** |  | | | |
| Withdraw money from ATM | T | F | F | F |

**State transition testing**

In this technique we need to understand the state flow of the use case then come up with design flow diagram. This flow can help us derive how many flows and how many are of positive and negative. In order to do this we must have proper knowledge on the application.

**Use case testing**

Use case tests are designed to be more customer specific test. The test case is written in the form a user guide where the steps are written in a procedural format. This gives more insight on the test procedure as it talks about the actors, events n pre and post conditions in specific.

Eg: let’s consider a login scenario into an application

**Actor**

Actor is the person who performs that events

**Pre-condition**

Actor should own a good network provision

Actor should be on the login page to be able to enter credentials

**Events**

Steps are addressed as events, the events involved in this step are

Enter the username and password

Click on the login button

**Alternatives**

This says what other actions can be possible, when actor on that page.

The actor can look for help link or actor can try signing up if the account is not create yet.

**Exceptions**

These are the problems that user might encounter during the execution of events

Application might be throwing an error upon entering the credentials

After the clicking the login button, occurrence of any HTTP error

**Post- actions**

These are the subsequent actions executed after the completion of events.

Logging out from the application

Performing any business functionality with in the application

**Conclusion**

Measuring test coverage metrics is quite essential as we need determine how much functionality of the application is being tested. Basis the analysis we can consider ramping up the scope by bringing possible infrastructure capabilities to the environment. Test coverage also helps us avoiding the unreachable code during development phase by checking the possible statement and decision coverage and during testing phase it defines the test scenario and use case flow coverage.

Hope there are some takeaways from the blog, keep reading ☺