UNIT 4

SYSTEM TESTING

CLASS TESTING – MUTATION TESTING

Class Testing

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-unit testing, small "units", or modules of the software, are tested separately with focus on testing the code of that module. In higher, order testing (e.g, acceptance testing), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behavior of the system.
As information systems are becoming more complex, the object-oriented paradigm is gaining popularity because of its benefits in analysis, design, and coding. Conventional testing methods cannot be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message, passing, polymorphism, concurrency, etc.
Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these approaches.

According to Davis the dependencies occurring in conventional systems are:

- Data dependencies between variables
- Calling dependencies between modules
- Functional dependencies between a module and the variable it computes
- Definitional dependencies between a variable and its types.

But in Object-Oriented systems there are following additional dependencies:

- Class to class dependencies
- Class to method dependencies
- Class to message dependencies
- Class to variable dependencies
- Method to variable dependencies
- Method to message dependencies
- Method to method dependencies

Issues in Testing Classes:

Additional testing techniques are, therefore, required to test these dependencies. Another issue of interest is that it is not possible to test the class dynamically, only its instances i.e, objects can be tested. Similarly, the concept of inheritance opens various issues e.g., if changes are made to a parent class or superclass, in a larger system of a class it will be difficult to test subclasses individually and isolate the error to one class.
In object-oriented programs, control flow is characterized by message passing among objects, and the control flow switches from one object to another by inter-object communication. Consequently, there is no control flow within a class like functions. This lack of sequential control flow within a class requires different approaches for testing. Furthermore, in a function, arguments passed to the function with global data determine the path of execution within the procedure. But, in an object, the state associated with the object also influences the path of execution, and methods of a class can communicate among themselves through this state because this state is persistent across invocations of methods. Hence, for testing objects, the state of an object has to play an important role.

Techniques of object-oriented testing are as follows:

1. **Fault Based Testing:**

   This type of checking permits for coming up with test cases supported the consumer specification or the code or both. It tries to identify possible faults (areas of design or code that may lead to errors.). For all of these faults, a test case is developed to “flush” the errors out. These tests also force each time of code to be executed.
   This method of testing does not find all types of errors. However, incorrect specification and interface errors can be missed. These types of errors can be uncovered by function testing in the traditional testing model. In the object-oriented model, interaction errors can be uncovered by scenario-based testing. This form of Object oriented-testing can only test against the client’s specifications, so interface errors are still missed.

2. **Class Testing Based on Method Testing:**

   This approach is the simplest approach to test classes. Each method of the class performs a well defined cohesive function and can, therefore, be related to unit testing of the traditional testing techniques. Therefore all the methods of a class can be involved at least once to test the class.

3. **Random Testing:**

   It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behavior of the categories

4. **Partition Testing:**

This methodology categorizes the inputs and outputs of a category so as to check them severely. This minimizes the number of cases that have to be designed.

5. **Scenario-based Testing:**

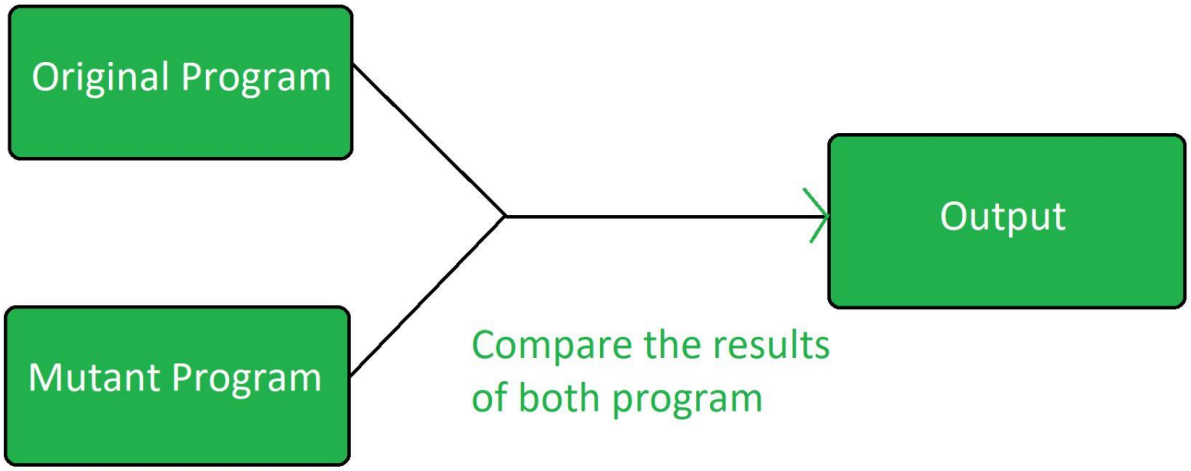
It primarily involves capturing the user actions then stimulating them to similar actions throughout the test.
These tests tend to search out interaction form of error.

Mutation Testing

Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

History of Mutation Testing:

Richard Lipton proposed the mutation testing in 1971 for the first time. Although high cost reduced the use of mutation testing but now it is widely used for languages such as Java and XML.



Mutation Testing is a White Box Testing.

Mutation testing can be applied to design models, specifications, databases, tests, and XML. It is a structural testing technique, which uses the structure of the code to guide the testing process. It can be described as the process of rewriting the source code in small ways in order to remove the redundancies in the source code.

Objective of Mutation Testing:

The objective of mutation testing is:

- To identify pieces of code that are not tested properly.
- To identify hidden defects that can't be detected using other testing methods.
- To discover new kinds of errors or bugs.
- To calculate the mutation score.
- To study error propagation and state infection in the program.
- To assess the quality of the test cases.

Types of Mutation Testing:

Mutation testing is basically of 3 types:

1. Value Mutations:

In this type of testing the values are changed to detect errors in the program. Basically a small value is changed to a larger value or a larger value is changed to a smaller value. In this testing basically constants are changed.

Example:

Initial Code:

```
int mod = 1000000007;
int a = 12345678;
int b = 98765432;
int c = (a + b) % mod;
```

Changed Code:

```
int mod = 1007;
int a = 12345678;
int b = 98765432;
int c = (a + b) % mod;
```

2. Decision Mutations:

In decisions mutations are logical or arithmetic operators are changed to detect errors in the program.

Example:

Initial Code:

```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:

```
if(a > b)
 c = 10;
```

```
else
 c = 20;
```

3. Statement Mutations:

In statement mutations a statement is deleted or it is replaces by some other statement.

Example:

Initial Code:

```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:

```
if(a < b)
 d = 10;
else
 d = 20;
```

Tools used for Mutation Testing :

- Judy
- Jester
- Jumble
- PIT
- MuClipse.

Advantages of Mutation Testing:

- It brings a good level of error detection in the program.
- It discovers ambiguities in the source code.
- It finds and solves the issues of loopholes in the program.
- It helps the testers to write or automate the better test cases.
- It provides more efficient programming source code.

Disadvantages of Mutation Testing:

- It is highly costly and time-consuming.
- It is not able for Black Box Testing.
- Some, mutations are complex and hence it is difficult to implement or run against various test cases.
- Here, the team members who are performing the tests should have good programming knowledge.
- Selection of correct automation tool is important to test the programs.