Lecture Notes

Unit Testing, TDD and Refactoring

Welcome to session on Unit Testing.

As you have been through the life cycle of Software Development, this module is about the essential part of the Software Life Cycle which comes after the development, that is testing.

Unit Testing is a level of software testing where individual units/ components of a software are tested.

In this Session, the following fundamentals of Software Unit Testing had been covered:

- What is a Software Unit
- What are Unit Test Cases
- Characteristics of Good Unit Test Cases
- Testing in JUnit
- Tags and Assertions in JUnit

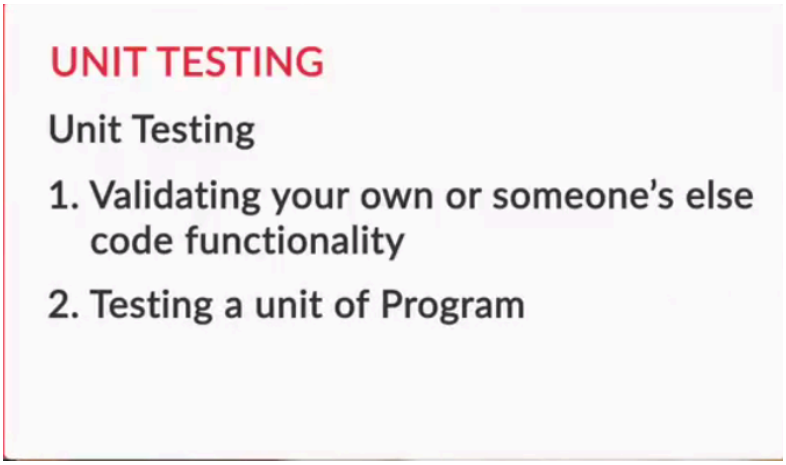
What is Unit Testing and What is a Software Unit

The term Unit Testing comprises of two words: Unit and Testing. Testing as you know is validating your own or someone else's code to check for the functionality.

A unit in Software Testing can be:

- A method
- A sequence of methods
- A class
- A sequence of classes

Any method that is public facing or deals with the outside world data can be thought of as a Software Unit.

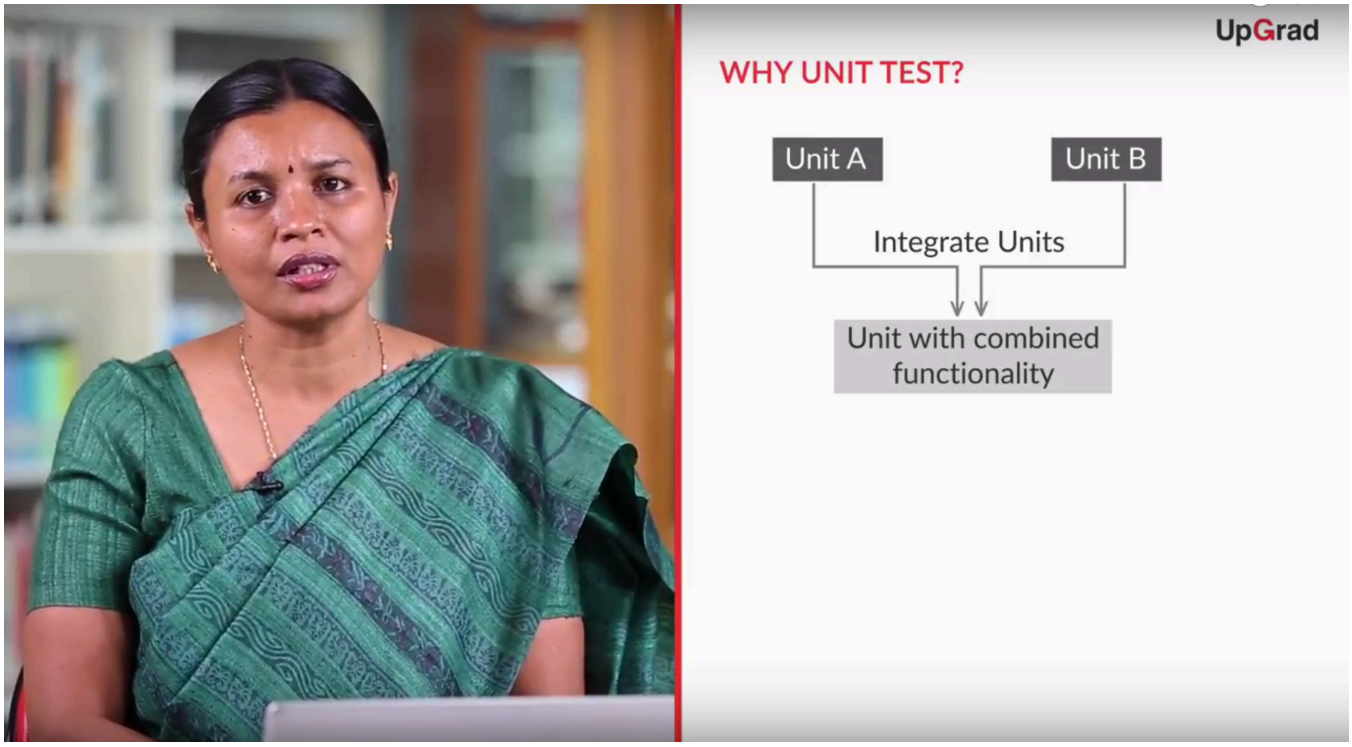


A UNIT OF PROGRAM

1. Method
2. Class
3. Sequence of Classes

Why Unit Testing

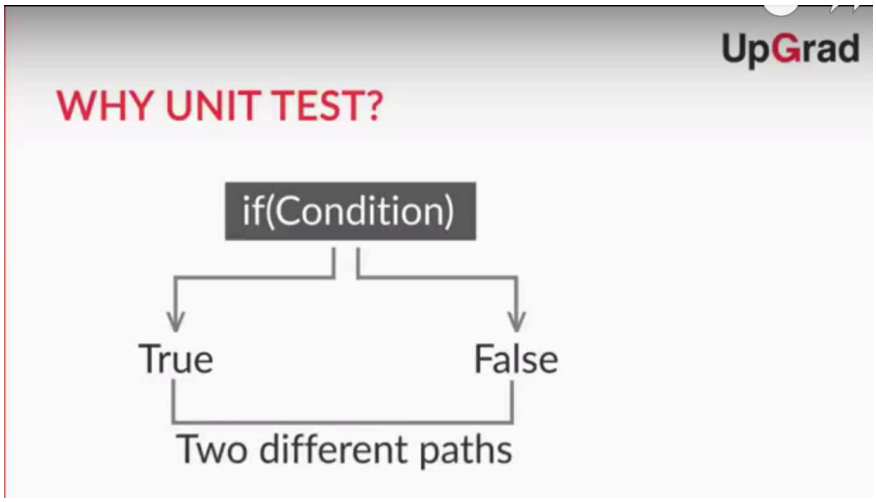
The certain question to ask here is why do we unit test, what is the benefit that it provides us with.



Let's say two teammates are working on a collaboration project and each of them is working on their individual units that needs to be combined together in the end to provide the desired functionality, then it is obviously going to save the overhead if the two individuals have been independently tested so as to simplify the process of integration of both the units. So, here Unit Testing plays a key role.

In Unit Testing, the important thing to check is that we consider each and every possible path a program might take, so as to give a certain assurance that the software would perform absolutely fine even under boundary conditions.

Let's say in case of an if condition, the tester needs to check for both if the condition is true and if the condition is false, so as to be certain whether the code is ready to be deployed or not.

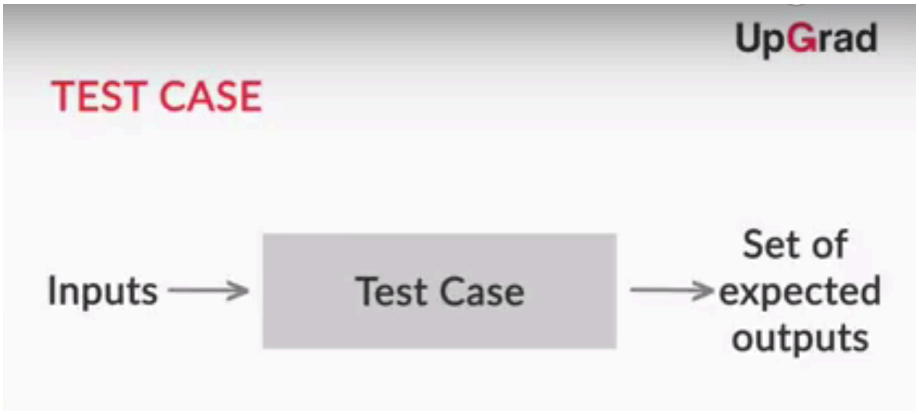


And in case of a for statement, we need to consider the end points of the loop during testing because this is where the unexpected behaviour happens.

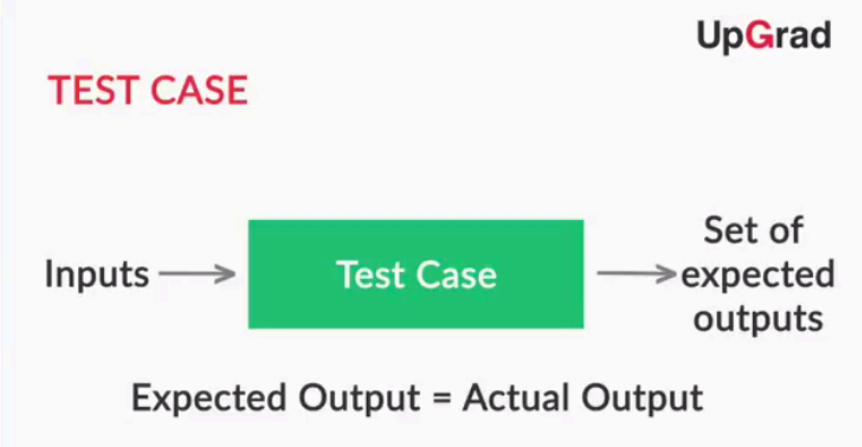
Now, it might be clear that to check if the desired functionality of the software is being fulfiled or not, we employ unit testing to test for the various units of the program.

Unit Test Cases

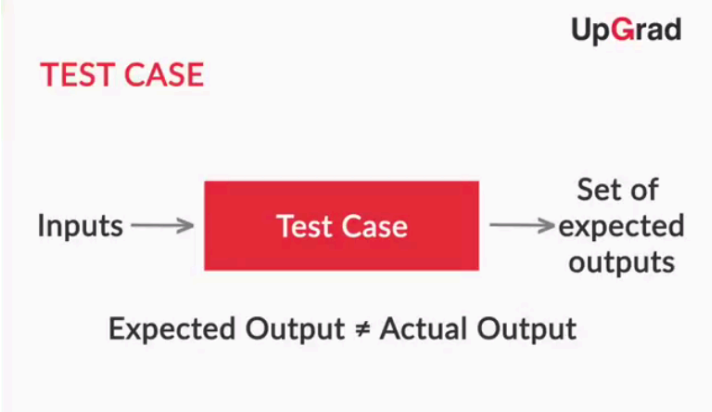
A test case can be thought of as a box which takes in inputs and gives a set of expected outputs which is then matched with the actual output of the code under test.



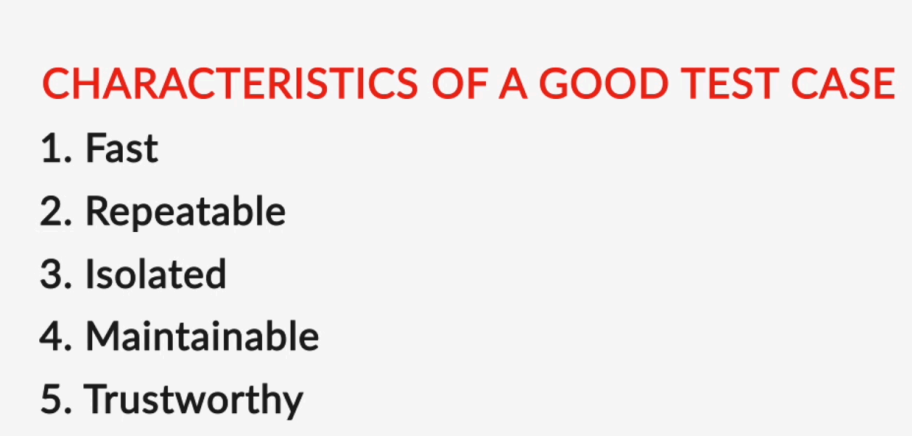
If the expected output matches the actual output, the test case is said to have passed.



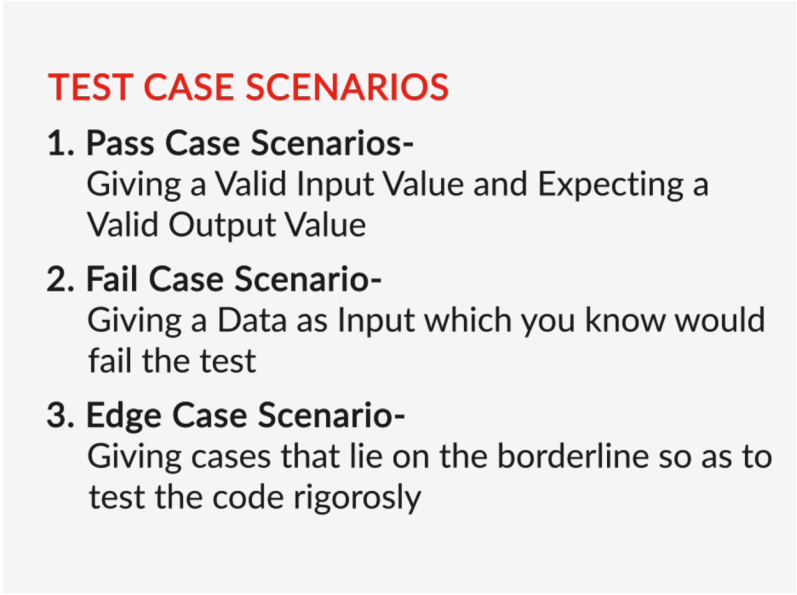
But if the expected output doesn't match the actual output, the test case is said to have failed.



Test Cases are the backbone of Unit Testing and creating good test cases is a really healthy practice in the process of Unit Testing. So, to create Good Unit Test cases, following factors should be taken into consideration:



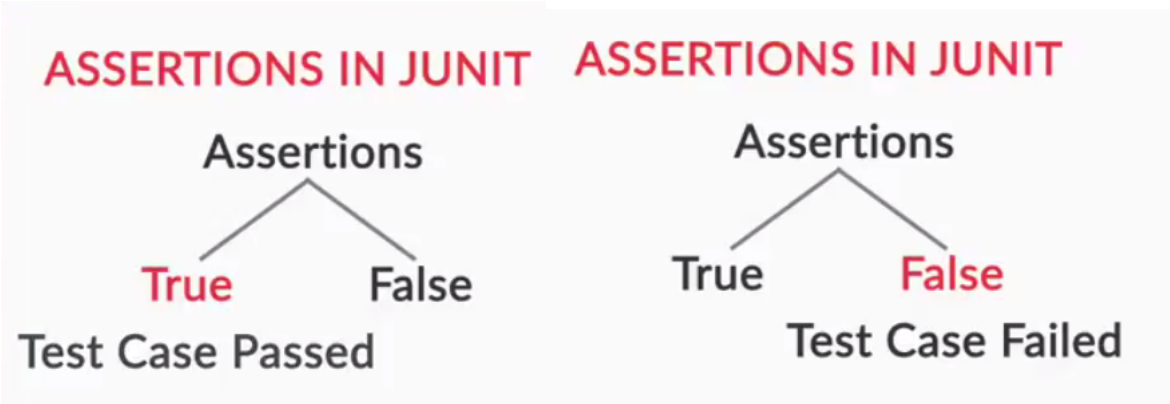
While testing a software, the test cases checks for certain scenarios or it can be said that the test cases are written to check for the following scenarios:



Assertions in Unit Testing

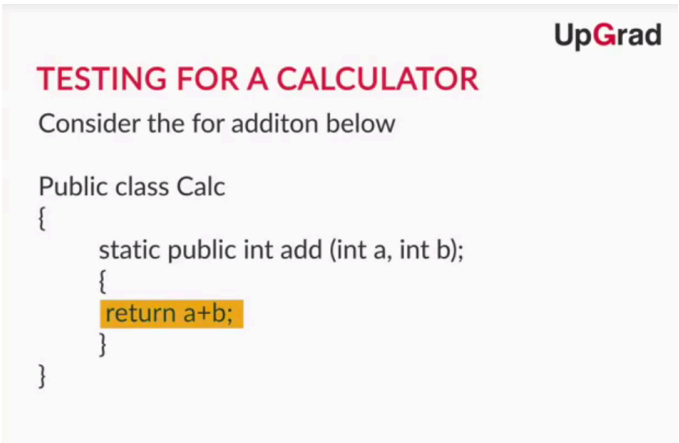
Now, you know about unit test cases, the obvious thing to ask is how to write test cases and determine whether they pass or fail, so we make use of methods known as Assertions.

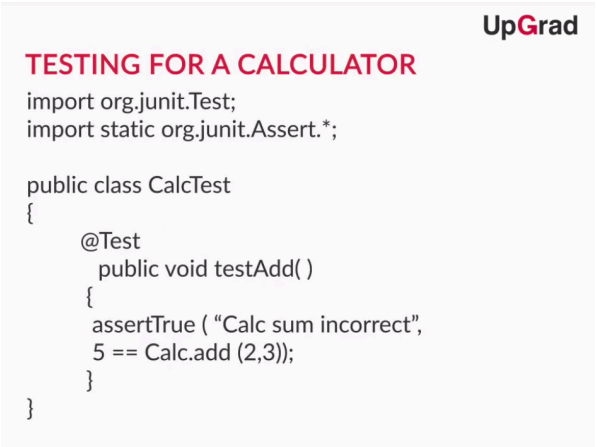
Assertions in Unit Testing are the methods which are used to determine the pass or fail status of a unit test case.



Assertions are the methods which are used to check for test case passing/failing behavior. If the result of assertion comes out to be true, then the test case is believed to be passed and if the result of assertion is false, then the test case is believed to be failed.

Underneath you see a sample code for a calculator which does addition of two numbers and there is also a test method written to check if the calculator program is working as it should or not.





The assertion assertTrue() checks if the boolean condition that has been passed to it is true or not, if it is true, then the assertion returns true, else the assertion returns false.

There are mainly four types of assertions that we have used.

- **AssertEquals():** Takes in two parameters and passes if the objects passed to it are equal.
- **AssertNull():** Takes in a single paramter and passes if the object passed to it is null.
- **AssertNotNull():** Takes in a single paramter and passes if the object passed to it is not null or something is contained by the object.
- **AssertTrue():** Takes in a boolean object and passes if the object value is true, else it fails.

Assertions are one of the A's of Unit Testing. Unit Testing is a process which consists of three A's

The three A's or AAA of unit testing stand for:

Arrange: This is the first step of a unit test application. Here we will arrange the test, in other words we will do the necessary setup of the test. For example, to perform the test we need to create an object of the targeted class.

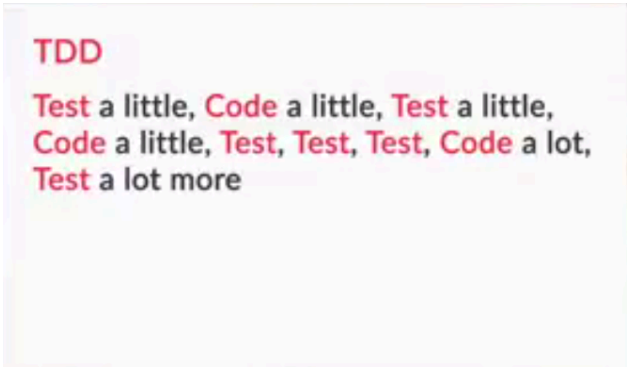
Act: This is the middle step of a unit step application. In this step we will execute the test. In other words we will do the actual unit testing and the result will be obtained from the test application. Basically we will call the targeted function in this step using the object that we created in the previous step.

Assert: This is the last step of a unit test application. In this step we will check and verify the returned result with expected results. Use of Assertions is in the Assert part of Unit Testing.

```
[TestClass]
public class UnitTest
{
        [TestMethod]
        public void TestMethod()
        {
        //Arrange test
                testClass objtest = new testClass();
                Boolean result;

        //Act test
                result = objtest.testFunction();

        //Assert test
                Assertions.assertEqual(true, result);

        }
}
```

Test Driven Development

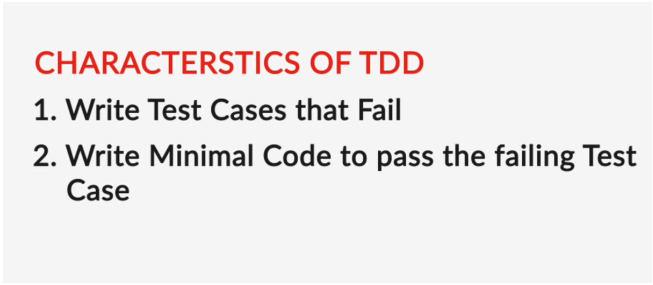
Test Driven Development or TDD as we call it is a process of software development in which testing and coding go hand in hand or in other words, we could say that in TDD, there is equal importance given to testing as is given to coding.



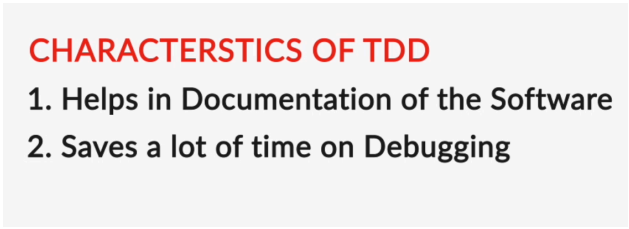
The above figure depicts TDD in a crisp manner, “We test a little, then code a little, test,code,test,code and the process goes on until we have a final deployable software”.

In TDD, development follows the order:

1. Write Test Cases that fail.
2. Write minimal code to pass that failing test case.
3. Refactor the code.



Benefits of TDD:

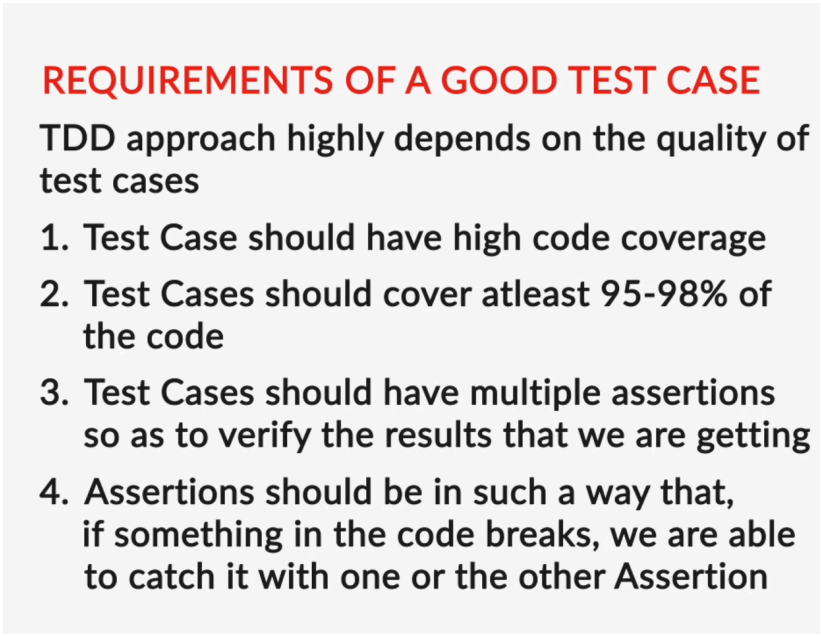


As we follow TDD, we sort of create a pseudo Documentation of the Software since we keep on writing test cases even before writing the code, so it helps in the latter stages of development since it already creates a set of test cases for the code. In this manner, it also saves the time on debugging later since the code that we write is already tested.

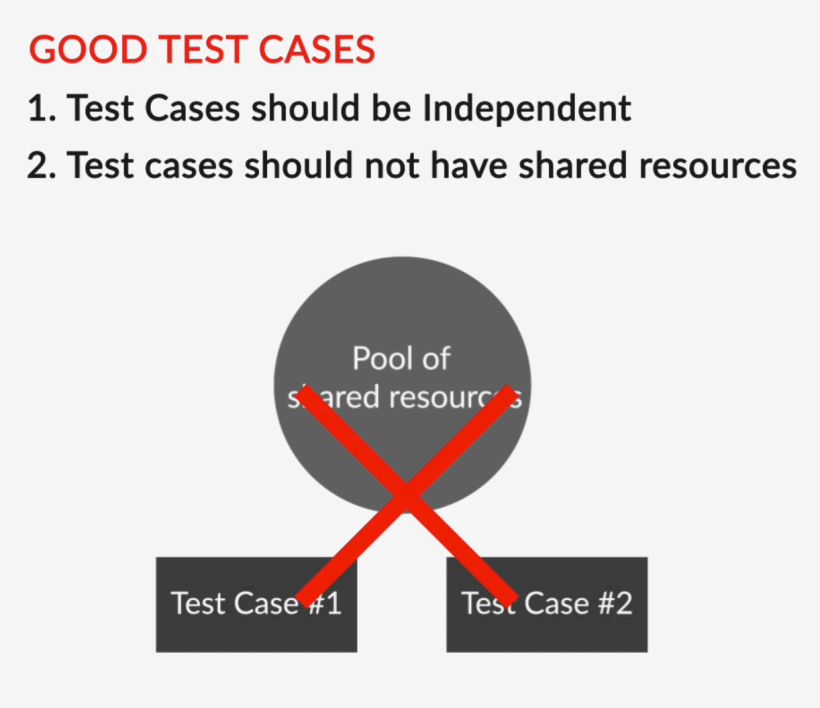
Requirements of a Good Test Case

Test cases are the integral part of the whole TDD process and a lot depends on the quality of test cases written.

So, the test cases should be written keeping these things in mind:



There are also certain things that we ought to keep in mind while writing the test cases that they shouldn't be part of the test case that we write so as to improve the quality of the test case.

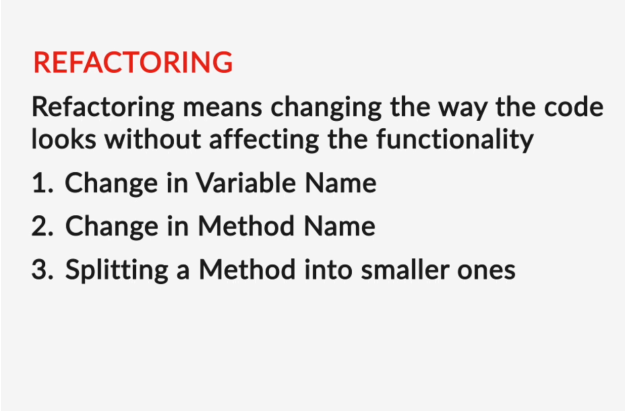


Since TDD is all about writing test cases and then coding to pass that test case, so the overall quality of test case needs to be maintained and the above points are what need to be considered while writing test cases.

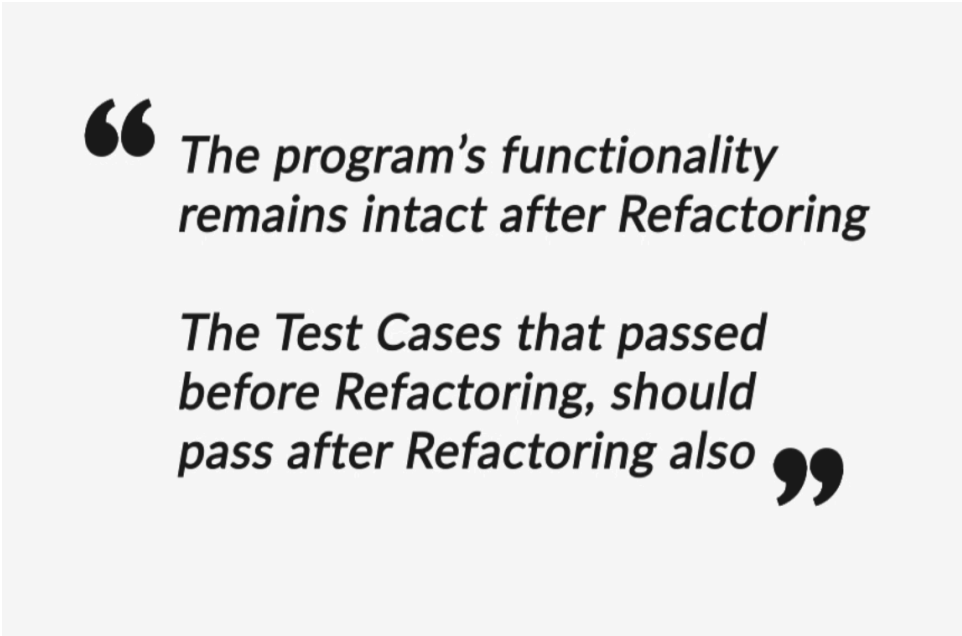
Refactoring

Refactoring is a term that you saw as being a step in the process of TDD. In this TDD, we actually end up writing bad code or code that smells. So to improve the quality of code that we write, we do Refactoring.

Refactoring in simple terms means changing the way the code looks without affecting the functionality of the code.

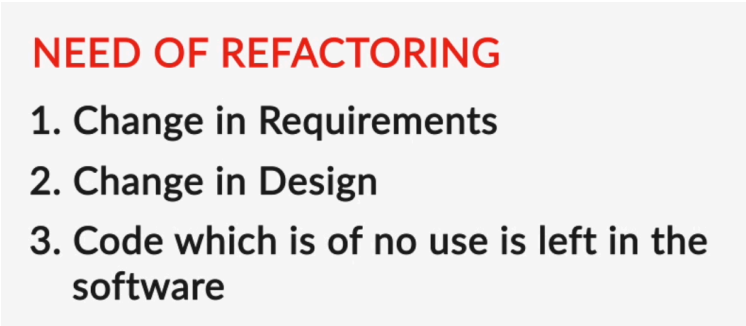


An important thing to keep in mind while employing Refactoring is that:



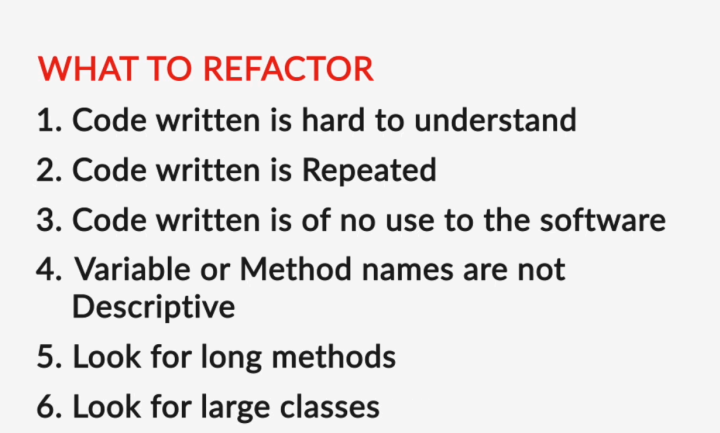
Need of Refactoring and What to Refactor

An interesting thing to ask here is what is the need of Refactoring or why do need Refactoring at all. The following figure depicts where exactly do we find the need of Refactoring:

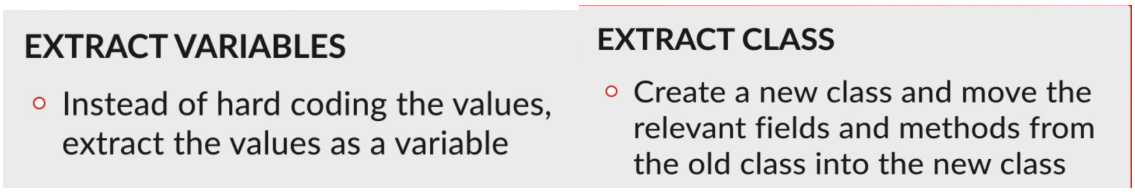


There are certain cases in software companies when there is a change in requirements of the client or there is change in the design of the software, there we need Refactoring. Sometimes, there is some code which is of no use left in the software, there also we need to remove that extra piece of code which is yet another way of Refactoring.

Now, the next thing is what to Refactor which is shown in the below figure:



You saw two particular ways to Refactor namely: Extract Variable and Extract Class.



When a certain value let's say 100 is being used at various places in the code to depict a maximum value, so rather than putting 100 everywhere, we can replace 100 with a variable let's say temp_Max=100 and use that temp_Max everywhere in the code, so this practice of replacing the value with a variable is what is known as Extract Variable.

Assume a class let's say calculator which consists of methods such as add, subtract, multiply and divide. Apart from these basic operations, the class also consists of certain scientific operations such as log, sqrt,etc. which can be realised as part of a scientific calculator, so we can move out these methods from the calculator class since they would be irrelevant to a basic calculator, this process is known as Extract Class Refactoring.

This brings an end to the module of 'Unit Testing,TDD and Refactoring'.