Experiment 5:

Obj: Calculate the mutation score of the following programs using jumble Tool:

5.1: Write a program that calculates the area and perimeter of the circle

5.2: Write a program which read the first name and last name from console and matching with expected result

5.3: Write a program that takes three double numbers from the java console representing, respectively, the three coefficients a,b, and c of a quadratic equation.

Description:

Mutation testing is a white box method in software testing where we insert errors purposely into a program (under test) to verify whether the existing test case can detect the error or not. In this testing, the mutant of the program is created by making some modifications to the original program.

The primary objective of mutation testing is to check whether each mutant created an output, which means that it is different from the output of the original program. We will make slight modifications in the mutant program because if we change it on a massive scale than it will affect the overall plan.

What is mutation?

The mutation is a small modification in a program; these minor modifications are planned to typical low-level errors which are happened at the time of coding process.

Types of mutation testing

Mutation testing can be classified into three parts, which are as follows:

- Decision mutations
- value mutations
- Statement mutations

Decision mutations

In this type of mutation testing, we will check the design errors. And here, we will do the modification in arithmetic and logical operator to detect the errors in the program.

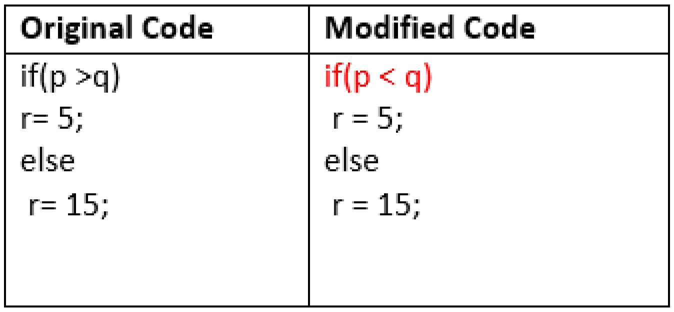
Like if we do the following changes in arithmetic operators:

- plus(+)→ minus(-)
- asterisk(*)→ double asterisk(**)
- plus(+)→incremental operator(i++)

Like if we do the following changes in logical operators

- Exchange P **>** → P**<**, OR P**>=**

Now, let see one example for our better understanding:



Original Code	**Modified Code**

if(p >q)
r= 5;
else
 r= 15;
	if(p < q)
	 r = 5;
	else
	 r = 15;

Value mutations

In this, the values will modify to identify the errors in the program, and generally, we will change the following:

- Small value à higher value
- Higher value àSmall value.

For Example:

Original Code	Modified Code

int add =9000008;
int p = 65432;
int q =12345;
int r = (p+ q);
	int mod = 9008;
	int p = 65432;
	int q =12345;
	int r= (p + q);

Statement Mutations

Statement mutations means that we can do the modifications into the statements by removing or replacing the line as we see in the below example:

Original Code	Modified Code

if(p * q)
 r = 15;
else
 r = 25;
	if(p* q)
	 s = 15;
	else
	 s = 25;

In the above case, we have replaced the statement r=15 by s=15, and r=25 by s=25.

Introduction to Jumble Tool :

The primary entry point to Jumble is a single Java class that takes as parameters a class to be mutation tested and one or more JUnit test classes. The output is a simple textual summary of running the tests. This is in the style of JUnit test output and includes an overall score of how many mutations were successfully caught as well as details about those mutations that are not. This includes the source line and the mutation performed. Variants of the output include a version compatible with Java development in emacs where it is possible to click on the line and go to the source line containing the mutation point. A plugin for the Eclipse IDE [10] has been provided. Currently it permits only jumbling of a single class. When running the mutation tests a separate JVM [11] is used to prevent runaway or system-exiting code disturbing the testing process. It is given a number which specifies the mutation to start at and it continues until all mutations have been tested or some catastrophic failure occurs. It has been very important to separate the test execution in this way for reasons discussed below. If a failure occurs in one mutation then the child JVM can be restarted and testing continues with the next mutation. Before running the mutation tests, a full run is done of the unit tests both to ensure that they all pass, to ensure that there are no environmental problems when running the tests in this way and to collect timing information to later detect mutations that lead to infinite loops. At Reel Two, Jumble is integrated with an internal continual integration system, on several source code repositories. Every fifteen minutes this checks out all the source code, clean compiles it and then runs all the unit tests for packages that have been modified. It also places all modified classes on a queue to be mutation tested. After the unit testing has been done, Jumble is used to test classes until the fifteen minute time limit is exceeded. Overnight more time is dedicated to mutation testing so that any tests that have been accumulated during the day can be cleared. The results of the Jumble tests for a project are presented in a web interface.

5.1: Write a program that calculates the area and perimeter of the circle

Program Code:

```
import java.util.Scanner;
class CircleDemo
{
static Scanner sc= new Scanner (System.in);
public static void main(String args[])
{
int a,b;
System.out.print("Enter the radius of Circle: ");
double radius = sc.nextDouble();
double area = Math.PI * (radius * radius);
System.out.print("The area of the Circle is: "+area);
double perimeter =Math.PI *2 * radius;
System.out.print(" The perimeter of circle is: " +perimeter);
}
}
```

5.2: Write a program which read the first name and last name from console and matching with expected result

```
import java.util.Scanner;
public class FullNameString {
public static void main(String[] args) {
Scanner input = new Scanner(System.in);
String savedname= "Manish";
System.out.print("Enter first name :: ");
String firstName = input.nextLine();
System.out.print("Enter middle name :: ");
String middleName = input.nextLine();
System.out.print("Enter surname :: ");
String lastName = input.nextLine();
// Here StringBuffer is used to combine 3 strings into single
StringBuffer fullName = new StringBuffer();
fullName.append(firstName);
fullName.append(" "); // For space between names
fullName.append(middleName);
fullName.append(" "); // For space between names
fullName.append(lastName);
System.out.println("Hello, " + fullName);
if (savedname.equals(firstName))
{
System.out.println("your First name is Match");
}
else {
System.out.println("your First name is Not Match");
}
}
}
```

5.3: Write a program that takes three double numbers from the java console representing, respectively, the three coefficients a,b, and c of a quadratic equation.

```
 package mypackage;
import java.util.Scanner;
public class QuadraticEquationExample1
{
public static void main(String[] Strings)
{
Scanner input = new Scanner(System.in);
System.out.print("Enter the value of a: ");
double a = input.nextDouble();
System.out.print("Enter the value of b: ");
double b = input.nextDouble();
System.out.print("Enter the value of c: ");
double c = input.nextDouble();
double d= b * b - 4.0 * a * c;
if (d> 0.0)
{
double r1 = (-b + Math.pow(d, 0.5)) / (2.0 * a);
double r2 = (-b - Math.pow(d, 0.5)) / (2.0 * a);
System.out.println("The roots are " + r1 + " and " + r2);
}
else if (d == 0.0)
{
double r1 = -b / (2.0 * a);
System.out.println("The root is " + r1);
}
else
{
System.out.println("Roots are not real.");
}
}
}
```