

# Batch: B1 Roll No.: 16010421119 Experiment No.:3

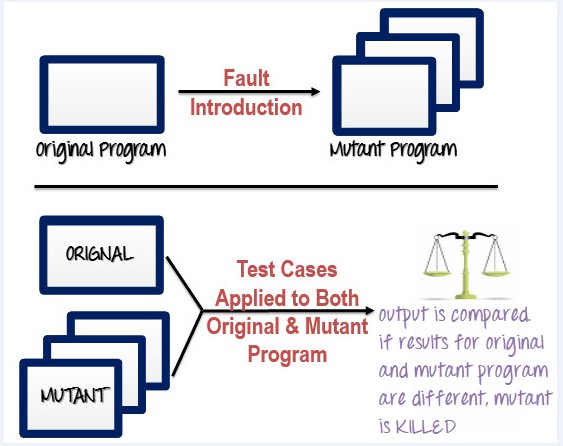
**Aim:** To perform mutation testing for measuring the test case coverage.

\_

**Resources needed:** Eclipse IDE, PIT Mutation Testing.

# Theory:

**Mutation Testing** is a type of software testing where we mutate (change) certain statements in the source code and check if the test cases are able to find the errors. It is a type of White Box Testing which is mainly used for Unit Testing.



# Mutation Testing Types:

* **Value Mutations:** An attempt to change the values to detect errors in the programs. We usually change one value to a much larger value or one value to a much smaller value. The most common strategy is to change the constants.
* **Decision Mutations:** The decisions/conditions are changed to check for the design errors. Typically, one changes the arithmetic operators to locate the defects and also we can consider mutating all relational operators and logical operators (AND, OR , NOT)
* **Statement Mutations:** Changes done to the statements by deleting or duplicating the line which might arise when a developer is copy pasting the code from somewhere else.

# How to Create Mutant Programs?

A mutation is nothing but a single syntactic change that is made to the program statement. Each mutant program should differ from the original program by one mutation.

Some of sample mutation operators:

* GOTO label replacement
* Return statement replacement
* Statement deletion
* Unary operator insertion (Like - and ++)
* Logical connector replacement
* Comparable array name replacement
* Removing of else part in the if-else statement
* Adding or replacement of operators
* Statement replacement by changing the data
* Data Modification for the variables
* Modification of data types in the program Example 1:

|  |  |
| --- | --- |
| **Original Program** | **Mutant Program** |
| If (x>y) | If**(**x<y**)** |
| Print "Hello" | Print "Hello" |
| Else | Else |
| Print "Hi" | Print "Hi" |

Example 2:

|  |  |
| --- | --- |
|  | Example- |
| Example- | If(x==y) |
| If(x>y) replace x and y values  If(5>y) replace x by constant 5 | We can replace == into >= and have mutant program as |
|  | If(x>=y) and inserting ++ in the statement |
|  | If(x==++y) |

# How to execute mutation testing?

Following are the steps to execute mutation testing:

**Step 1**: Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases.

**Step 2**: Test cases are applied to the original program and also to the mutant program. A [test](https://www.guru99.com/test-case.html) [case](https://www.guru99.com/test-case.html) should be adequate, and it is tweaked to detect faults in a program.

**Step 3**: Compare the results of original and mutant program.

**Step 4**: If the original program and mutant programs generate the different output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program.

**Step 5**: If the original program and mutant program generate same output, Mutant is kept alive. In such cases, more effective test cases need to be created that kill all mutants.

# Automation of Mutation Testing:

Mutation testing is extremely time consuming and complicated to execute manually. To speed up the process, it is advisable to go for automation tools. Automation tools reduce cost of testing as well.

List of tools available -

[Ninja Turtles](http://www.mutation-testing.net/)- .net mutation testing tool [Mutagenesis](https://github.com/padraic/mutagenesis)- [PHP](https://www.guru99.com/php-tutorials.html) mutation testing framework [Jester](http://jester.sourceforge.net/)- Mutation testing tool for Java

[PITest- Mutation testing](http://pitest.org/) tool for Java and the JVM

# Mutation Score:

The mutation score is defined as the percentage of killed mutants with the total number of mutants.

* Mutation Score = (Killed Mutants / Total number of Mutants) \* 100

Test cases are mutation adequate if the score is 100%. Experimental results have shown that mutation testing is an effective approach for the measuring the adequacy of the test cases. But, the main drawback is that the high cost of generating the mutants and executing each test case against that mutant program.

# Procedure:

Step 1: To download [PITest- Mutation Testing](http://pitest.org/)[ Eclipse-> help-> Market places -> type “PITest”-> select PITest from the list-> next]

Step 2: write the test cases and run [right click on the test case file-> run as-> PITest] Step 3: Observe the result and write additional test case to achieve 100% score.

# Results: (Document printout as per the format)

Write test cases and perform [PITest- Mutation Testing](http://pitest.org/) for 100% score and document the same.

# STQA.java

**package** stqa;

**public class stqa** {

**public** int calculateDiscount(double purchaseAmount, boolean isMember) {

**if** (purchaseAmount < 0) {

**throw** new IllegalArgumentException("Purchase amount cannot be negative");

}

**if** (purchaseAmount >= 1000) {

**if** (isMember) {

**return** 20;

} **else** {

**return** 10;

}

} **else if** (purchaseAmount >= 500) {

**if** (isMember) {

**return** 10;

} **else** {

**return** 5;

}

} **else if** (purchaseAmount >= 100) {

**if** (isMember) {

**return** 5;

} **else** {

**return** 0;

}

} **else** {

**return** 0;

}

}

}

**STQATEST.java**

**package** stqa;

**import** org.junit.Test;

**import static** org.junit.Assert.\*;

**public class stqaTest** {

stqa discountCalculator=**new** stqa();

@Test

**public void testCalculateDiscountHighPurchaseMember**() {

**int** discount = discountCalculator.calculateDiscount(1000,

**true**);

assertEquals(20, discount);

}

@Test

**public void testCalculateDiscountHighPurchaseNonMember**() {

**int** discount = discountCalculator.calculateDiscount(1000,

**false**);

assertEquals(10, discount);

}

@Test

**public void testCalculateDiscountJustBelowHighPurchaseMember**()

{

**int** discount =

discountCalculator.calculateDiscount(999.99, **true**);

assertEquals(10, discount);

}

@Test

# public void

**testCalculateDiscountJustBelowHighPurchaseNonMember**() {

**int** discount =

discountCalculator.calculateDiscount(999.99, **false**); assertEquals(5, discount);

}

@Test

# public void testCalculateDiscountMediumPurchaseMember() {

**int** discount = discountCalculator.calculateDiscount(500,

**true**);

}

assertEquals(10, discount);

@Test

# public void testCalculateDiscountMediumPurchaseNonMember() {

**int** discount = discountCalculator.calculateDiscount(500,

**false**);

}

assertEquals(5, discount);

@Test

# public void

**testCalculateDiscountJustBelowMediumPurchaseMember**() {

**int** discount =

discountCalculator.calculateDiscount(499.99, **true**); assertEquals(5, discount);

}

@Test

# public void

**testCalculateDiscountJustBelowMediumPurchaseNonMember**() {

**int** discount =

discountCalculator.calculateDiscount(499.99, **false**); assertEquals(0, discount);

}

@Test

# public void testCalculateDiscountLowPurchaseMember() {

**int** discount = discountCalculator.calculateDiscount(100,

**true**);

}

assertEquals(5, discount);

@Test

# public void testCalculateDiscountLowPurchaseNonMember() {

**int** discount = discountCalculator.calculateDiscount(100,

**false**);

}

assertEquals(0, discount);

@Test

# public void testCalculateDiscountBelowMinimum() {

**int** discount = discountCalculator.calculateDiscount(0,

**true**);

}

assertEquals(0, discount);

@Test

# public void testCalculateDiscountNegativePurchase() {

**try** {

discountCalculator.calculateDiscount(-10, **true**);

} **catch** (IllegalArgumentException e) {

assertEquals("Purchase amount cannot be negative", e.getMessage());

}

}

@Test

# public void testCalculateDiscountLargePurchase() {

**int** discount =

discountCalculator.calculateDiscount(1\_000\_000, **true**); assertEquals(20, discount);

}

@Test

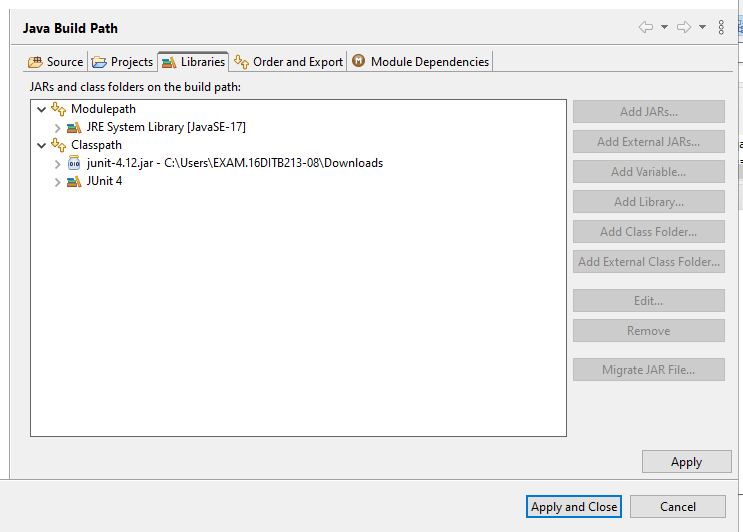
**public void testCalculateDiscountLargePurchaseNonMember**() {

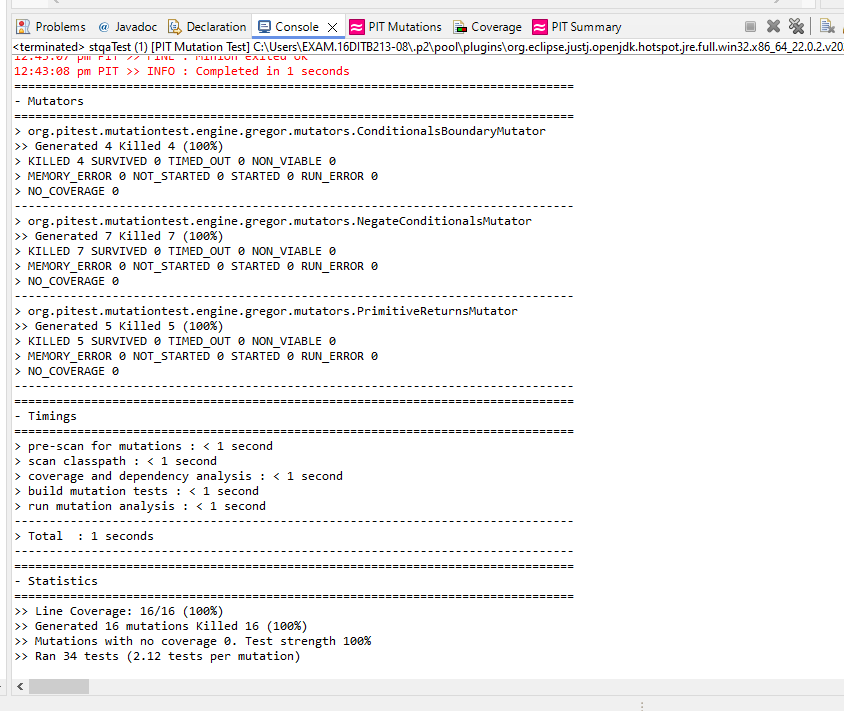
**int** discount =

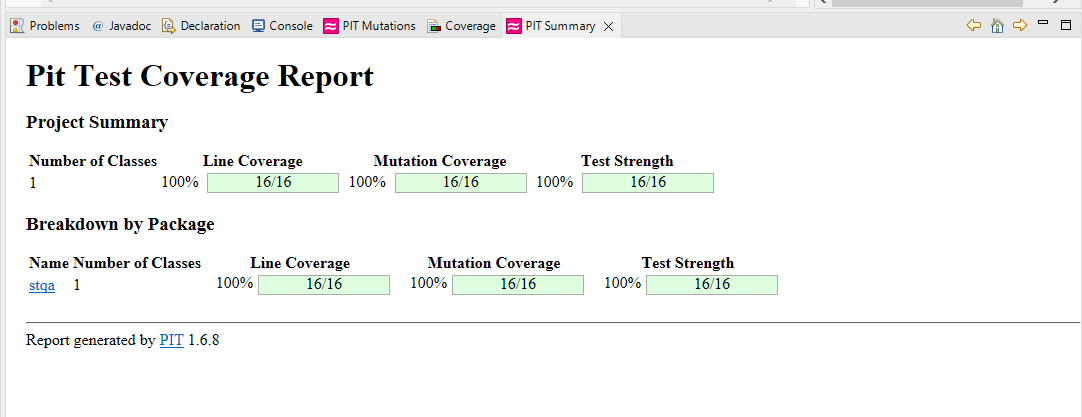
discountCalculator.calculateDiscount(1\_000\_000, **false**); assertEquals(10, discount);

}

}







# Questions:

1. What is equivalent Mutant? Explain with example.

Equivalent mutants are a concept from software testing, particularly in mutation testing. Mutation testing involves modifying a program's code in small ways (creating "mutants") to

see if the existing test cases can detect the changes. The idea is to test the quality of the test cases: if a test case can detect a mutant (i.e., it fails when the mutant code is run), it is considered strong.

An equivalent mutant is a mutant that, despite the changes made to the code, still behaves the same as the original code for all possible inputs. In other words, the mutant does not introduce any new behavior that the test cases can detect. Because of this, the test cases will not fail when run against an equivalent mutant, even though the code has been altered.

Example 1: Addition Operation

Imagine you have a function that adds two numbers. If you switch the order of the numbers, the result will still be the same because addition works the same way regardless of the order. This means the change doesn’t introduce any new behavior, making it an equivalent mutant.

Example 2: Multiplication by 1

Consider a function that multiplies two numbers. If you modify it to multiply one of the numbers by 1 before doing the original multiplication, the outcome will remain unchanged. Since multiplying by 1 doesn't affect the result, this modification also creates an equivalent mutant.

Example 3: Redundant Check

Imagine a function that checks if a number is positive. If you add an extra condition to check if the number is zero, but this condition doesn’t affect the original logic, the function’s behavior stays the same. This makes the change an equivalent mutant.

**Outcomes: CO2: Demonstrate designing and execution of test cases using testing techniques.**

# Conclusion: (Conclusion to be based on outcomes)

In completing this experiment, I successfully performed mutation testing for measuring the test case coverage using Java programs on Eclipse IDE.

# Grade: AA / AB / BB / BC / CC / CD /DD Signature of faculty in-charge with date

**Reference Websites:**

* 1. [Mutation Tasting Tutorial](https://www.tutorialspoint.com/testng/) <https://www.tutorialspoint.com/software_testing_dictionary/mutation_testing.htm> <https://www.guru99.com/mutation-testing.html>
  2. Mutation Testing

<https://www.youtube.com/watch?v=1DvJeMmKrUE>

* 1. [Mutation Testing With PIT](https://www.youtube.com/playlist?list=PLSRWfK9C0fXWHD-u-3VYKZm-dcNRMAWqp) <https://www.youtube.com/watch?v=3Al1tHE8z64>