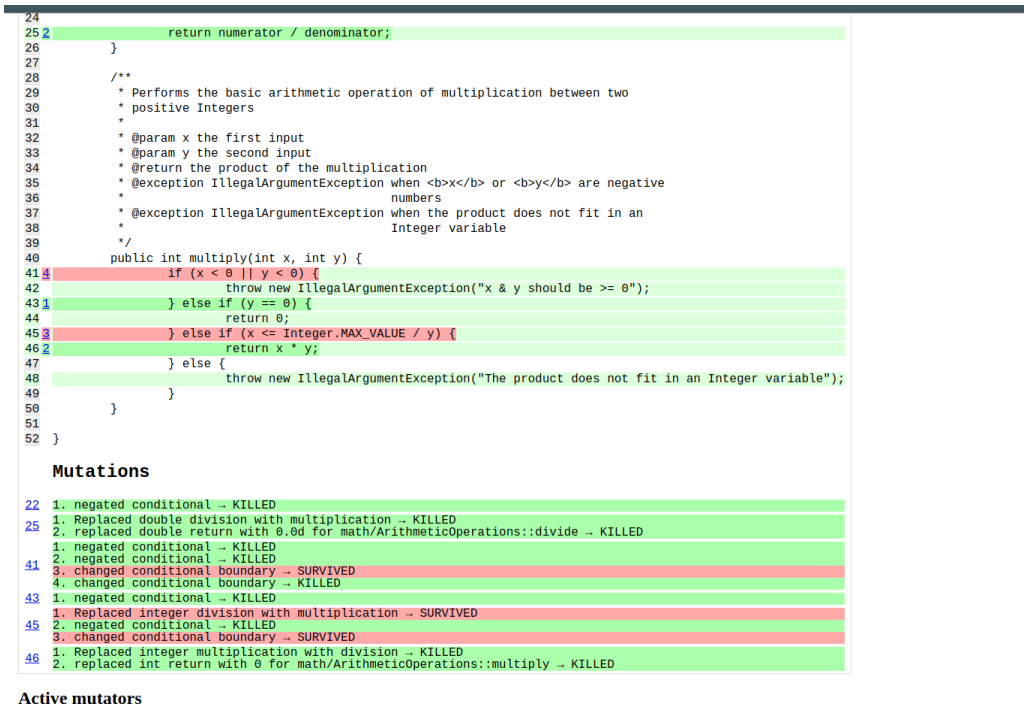
PIT Test Coverage Report

Multiply Method in ArithmeticOperations class:



```
24
25 2            return numerator / denominator;
26      }
27
28      /**
29       * Performs the basic arithmetic operation of multiplication between two
30       * positive Integers
31       *
32       * @param x the first input
33       * @param y the second input
34       * @return the product of the multiplication
35       * @exception IllegalArgumentException when <b>x</b> or <b>y</b> are negative
36       *                                     numbers
37       * @exception IllegalArgumentException when the product does not fit in an
38       *                                     Integer variable
39       */
40      public int multiply(int x, int y) {
41 4            if (x < 0 || y < 0) {
42                  throw new IllegalArgumentException("x & y should be >= 0");
43 1            } else if (y == 0) {
44                  return 0;
45 3            } else if (x <= Integer.MAX_VALUE / y) {
46 2                return x * y;
47            } else {
48                  throw new IllegalArgumentException("The product does not fit in an Integer variable");
49            }
50      }
51
52 }
```

Mutations

```
22  1. negated conditional → KILLED
25  1. Replaced double division with multiplication → KILLED
    2. replaced double return with 0.0d for math/ArithmeticOperations::divide → KILLED
41  1. negated conditional → KILLED
    2. negated conditional → KILLED
    3. changed conditional boundary → SURVIVED
    4. changed conditional boundary → KILLED
43  1. negated conditional → KILLED
45  1. Replaced integer division with multiplication → SURVIVED
    2. negated conditional → KILLED
    3. changed conditional boundary → SURVIVED
46  1. Replaced integer multiplication with division → KILLED
    2. replaced int return with 0 for math/ArithmeticOperations::multiply → KILLED
```

Active mutators

The if(x<0 || y<0) mutation got survived because there was no testcase to verify the boundary value input of x==0 or y==0, as a result the mutant x<=0 || y<=0 survived. So I added a testcase of x==0 that caused the mutant to fail and get killed in mutation test.

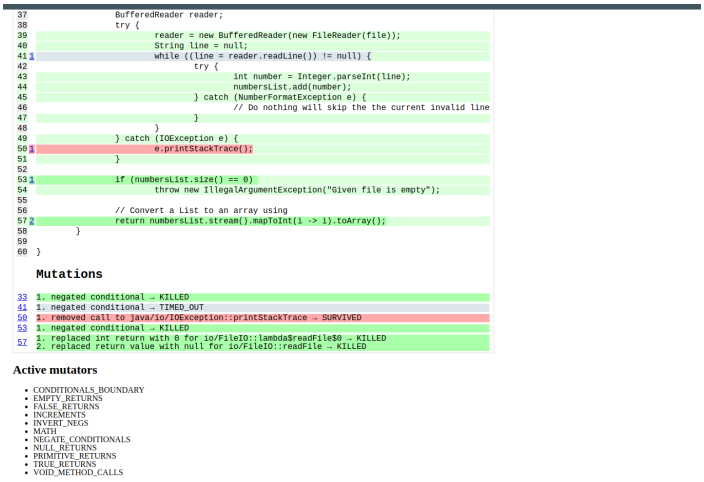
```
@Test  tashrif-007
public void multiply_with_zero_first_parameter() {
    int actual = (new ArithmeticOperations()).multiply( x: 0,  y: 5);
    assertEquals( expected: 0, actual);
}
```

The **else if(x<=Integer.MAX_VALUE/y)** was mutated to x<Integer.MAX_VALUE/y, so there was no test case for checking x = Integer.MAX_VALUE/y which cause the mutant to survive. I added the following test to kill it:

```java
@Test  tashrif-007
public void multiply_boundary_case_max_value() {
    int y = 2;
    int x = Integer.MAX_VALUE / y;
    int expected = x * y;
    int actual = (new ArithmeticOperations()).multiply(x, y);
    assertEquals(expected, actual);
}
```

Now it has killed all mutants having 100% Coverage in this class.

FileIO Class:



```
37              BufferedReader reader;
38              try {
39                      reader = new BufferedReader(new FileReader(file));
40                      String line = null;
41 1                    while ((line = reader.readLine()) != null) {
42                              try {
43                                      int number = Integer.parseInt(line);
44                                      numbersList.add(number);
45                              } catch (NumberFormatException e) {
46                                      // Do nothing will skip the the current invalid line
47                              }
48                      }
49              } catch (IOException e) {
50 1                    e.printStackTrace();
51              }
52
53 1            if (numbersList.size() == 0)
54                      throw new IllegalArgumentException("Given file is empty");
55
56              // Convert a List to an array using
57 2            return numbersList.stream().mapToInt(i -> i).toArray();
58      }
59
60 }
```

Mutations

```
33 1. negated conditional → KILLED
41 1. negated conditional → TIMED_OUT
50 1. removed call to java/io/IOException::printStackTrace → SURVIVED
53 1. negated conditional → KILLED
57 1. replaced int return with 0 for io/FileIO::lambda$readFile$0 → KILLED
   2. replaced return value with null for io/FileIO::readFile → KILLED
```

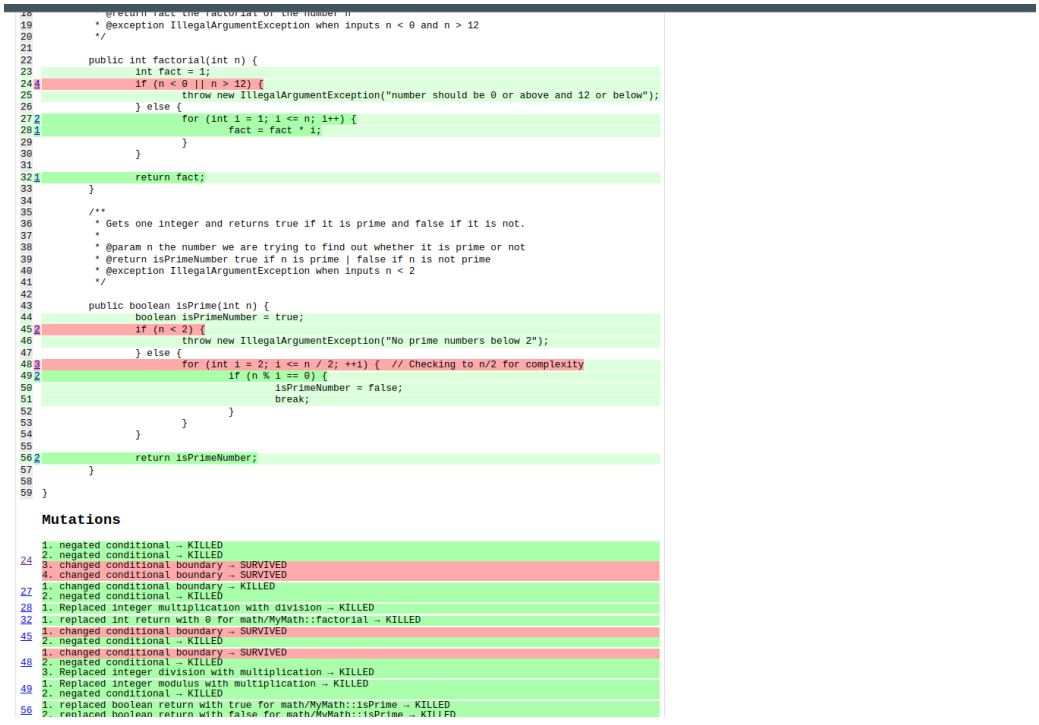
Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

The mutant survived because **e.printStackTrace()** on line 50 is a void method that only produces side effects (printing to stderr) without affecting the method's return value or exception behavior. Previous tests only verified functional outcomes (return values and exceptions) but ignored whether the stack trace was actually printed. The testIOExceptionHandling() test killed this mutant by deliberately triggering an **IOException** (reading a directory as a file), redirecting System.err to capture the output, and then asserting that "IOException" appears in the captured stderr content - thus verifying that printStackTrace() was actually executed and detecting when the mutation removes this call.

```java
@Test  new *
public void testIOExceptionHandling() {
    File dir = new File( pathname: "src/test/resources/io_exception_dir");
    if (!dir.exists()) {
        assertTrue(dir.mkdirs());  // create as a directory
    }
    PrintStream originalErr = System.err;
    ByteArrayOutputStream errContent = new ByteArrayOutputStream();
    System.setErr(new PrintStream(errContent));
    try {
        fileIO.readFile(dir.getAbsolutePath());
    } catch (Exception ignored) {
    } finally {
        System.setErr(originalErr);
        assertTrue(dir.delete());
    }
    String output = errContent.toString();
    assertTrue( message: "Should contain IOException in stack trace", output.contains("IOException"));
}
```

MyMath Class:



```
         * @exception IllegalArgumentException when inputs n < 0 and n > 12
         */

        public int factorial(int n) {
                int fact = 1;
24 4            if (n < 0 || n > 12) {
                        throw new IllegalArgumentException("number should be 0 or above and 12 or below");
                } else {
27 2                    for (int i = 1; i <= n; i++) {
28 1                            fact = fact * i;
                        }
                }

32 1            return fact;
        }

        /**
         * Gets one integer and returns true if it is prime and false if it is not.
         *
         * @param n the number we are trying to find out whether it is prime or not
         * @return isPrimeNumber true if n is prime | false if n is not prime
         * @exception IllegalArgumentException when inputs n < 2
         */

        public boolean isPrime(int n) {
                boolean isPrimeNumber = true;
45 2            if (n < 2) {
                        throw new IllegalArgumentException("No prime numbers below 2");
                } else {
48 3                    for (int i = 2; i <= n / 2; ++1) {  // Checking to n/2 for complexity
49 2                            if (n % i == 0) {
                                        isPrimeNumber = false;
                                        break;
                                }
                        }
                }

56 2            return isPrimeNumber;
        }
}
```

Mutations

```
24  1. negated conditional → KILLED
    2. negated conditional → KILLED
    3. changed conditional boundary → SURVIVED
    4. changed conditional boundary → SURVIVED
27  1. changed conditional boundary → KILLED
    2. negated conditional → KILLED
28  1. Replaced integer multiplication with division → KILLED
32  1. replaced int return with 0 for math/MyMath::factorial → KILLED
45  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
48  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
    3. Replaced integer division with multiplication → KILLED
49  1. Replaced integer modulus with multiplication → KILLED
    2. negated conditional → KILLED
56  1. replaced boolean return with true for math/MyMath::isPrime → KILLED
    2. replaced boolean return with false for math/MyMath::isPrime → KILLED
```

The n<0 || n>12 mutant was checking equality of boundaries which were not checked in the test case, so adding n==0 or n==12 in the test case killed this mutant.

```java
@Test  tashrif-007
public void factorialBoundaryZero() {
    assertEquals( expected: 1, (new MyMath()).factorial( n: 0));
}

@Test  tashrif-007
public void factorialBoundaryTwelve() {
    assertEquals( expected: 479001600, (new MyMath()).factorial( n: 12));
}
```

n<2 and the loop part mutation was fixed by adding test case n==2 and n==4. As for for (int i = 2; i <= n / 2; ++i), i is checked for 2 < 2 having false and prints 2 as prime which fails the test case and gets killed.

```java
@Test  tashrif-007
public void isPrimeFour() {
    assertEquals( expected: false, (new MyMath()).isPrime( n: 4));
}
```

So lastly the overall coverage for the classes are 100%.

Pit Test Coverage Report

Package Summary

math

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
3	100%	33/33	100%	35/35	100%	35/35

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
ArithmeticOperations.java	100%	11/11	100%	13/13	100%	13/13
ArrayOperations.java	100%	7/7	100%	5/5	100%	5/5
MyMath.java	100%	15/15	100%	17/17	100%	17/17