☰ **Exercise Overview** (0/0 complete) ▼

2.13. Mutation Testing Basics

2.13.1. Types of Mutants

The main purpose of writing test cases is to make sure that your solution code is working properly and is producing the behavior that you want. There are different mutators in mutation testing that allows you to check if your test suite is properly testing every element in your project solution code. This is a great way to make sure that your solution code is working according to your intended logic.

Another benefit of using mutation testing is to make sure your solution code has good design and abides by good programming practices. This means your solution code does not have unnecessary complexities, dead code (code that can never be executed), bad design, technical debts, etc.

There are different mutators that ensure different qualities of a programming project. For the programming projects in this course, we are going to use only two mutators to generate the following mutants:

1. Arithmetic Operation Mutant
2. Logical Expression Mutant (Remove Conditionals)

Don't worry, you will not have to set anything. If you have properly intalled the latest version of the Web-CAT Submission Plug-in, then the mutation testing plug-in in your Eclipse IDE should come with the desired two mutators set as default.

In the following sections, we will look into:

What these two mutators are.

How they work.

What feedback they produce

How to use the HINT in the feedback.

How to write test cases to improve mutation coverage.

Different examples that use these two mutators.

2.13.1.1. Arithmetic Operation Mutant

The arithmetic operation mutant checks if an arithmetic operation is being tested by your test suite. This mutator replaces an arithmetic operation with one of its members. The mutator is composed of 2 sub-mutators, AOD_1 and AOD_2, that mutate the operation to its first and second member respectively.

If the test suite detects the change i.e. mutanting the code results in a test failing, then the test suite passes. The terminology is that the test suite “killed the mutation”. This results in a higher mutation coverage score. When the test suite fails to detect the mutant, this results in a lower mutation coverage score. In such case, we need to write additional test case assertions to test for the arithmetic operation in question.

For example

```
int a = b + c;
```

will be mutated to

```
int a = b;        // replaced with first member
```

and to

```
int a = c;        // replaced with second member
```

Here, b``is the first member and ``c is the second member.

This way, mutation testing ensures that the arithmetic operator is tested for its intended behavior.

2.13.1.2. Example Code 1: Arithmetic Operation Mutant

As an example, we want to write a function that takes two numbers and returns the sum.

```
    public static int Addition(int num1, int num2) {
            int sum = 0;

            sum = num1 + num2;         // --> math operation

            return sum;
    }
```

Now if we execute mutation testing it will mutate the code as follows:

Replacing the arithmetic operation with first member:

```
    public static int Addition(int num1, int num2) {
            int sum = 0;

            sum = num1;         // --> math operation

            return sum;
    }
```

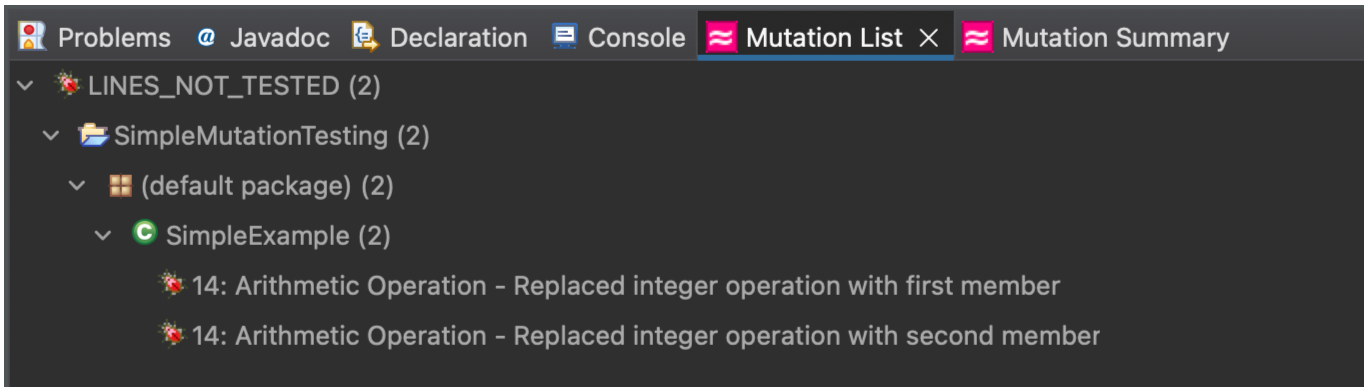
Replacing the arithmetic operation with second member:

```
    public static int Addition(int num1, int num2) {
            int sum = 0;

            sum = num2;          // --> math operation

            return sum;
    }
```

If we have no test cases execute this code, the test process will generate the following mutations in the LINES_NOT_TESTED group under the Mutations List tab. (The icons for unresolve mutants are shown as "red bugs".)



```
LINES_NOT_TESTED (2)
  SimpleMutationTesting (2)
    (default package) (2)
      SimpleExample (2)
        14: Arithmetic Operation - Replaced integer operation with first member
        14: Arithmetic Operation - Replaced integer operation with second member
```

Figure 2.13.1: Example Code 1: Arithmetic Operation Mutant without test cases

Here, we can see that the HINT suggests that we write test case assertions to test the arithmetic operation for intended behavior.

The following test case will kill both of these mutations:

```
    // testAddition tests for adding two numbers
    @Test
    public void testAddition() {
            // testing if 5+10 == 15
            assertEquals(15, SimpleExample.Addition(5, 10));
    }
```

If we run the mutation testing again then we will not get any of the previous mutations in the LINES_NOT_TESTED group under the Mutations List tab.

2.13.1.3. Logical Expression Mutant (Remove Conditionals)

The logical expression mutator (a.k.a. remove conditionals mutator) checks if a logical expression is properly tested by your test suite. This mutator replaces the logical expression with either TRUE or FALSE and then runs your test suite with the mutant.

For example replacing logical expression with TRUE condition:

```
if (a == b) {
// do something
}
```

will be mutated to

```
if (true) {
// do something
}
```

For example replacing logical expression with FALSE condition:

```
if (a == b) {
// do something
}
```

will be mutated to

```
if (false) {
// do something
}
```

If there is more than one logical expression then each expression will be mutated in separate runs of the test suite. The logical expression mutator also mutates the bytecode instructions for order checks (e.g. <=, >).

If there are more than one logical expression in the same statement, then the generated mutants will be in order of the logical expressions in the statement. Keep in mind, for multiple logical expressions, you must test each and every one of the expressions.

2.13.1.4. Example Code 2: Logical Expression Mutant (Remove Conditionals)

As an example, we want to write a function that takes a number and returns TRUE if the number is positive and FALSE if the number is zero or negative.

```
    public static boolean PositiveCheck(int number) {
            if (number > 0) {                                       // --> true or fals
                    return true;                                    // positive number
            }
            else {
                    return false;                                   // zero or negative
```

```
                        return false;                           // zero or negative
                }
        }
```

Executing mutation testing will mutate the code as follows.

Replacing the logical expression with TRUE:

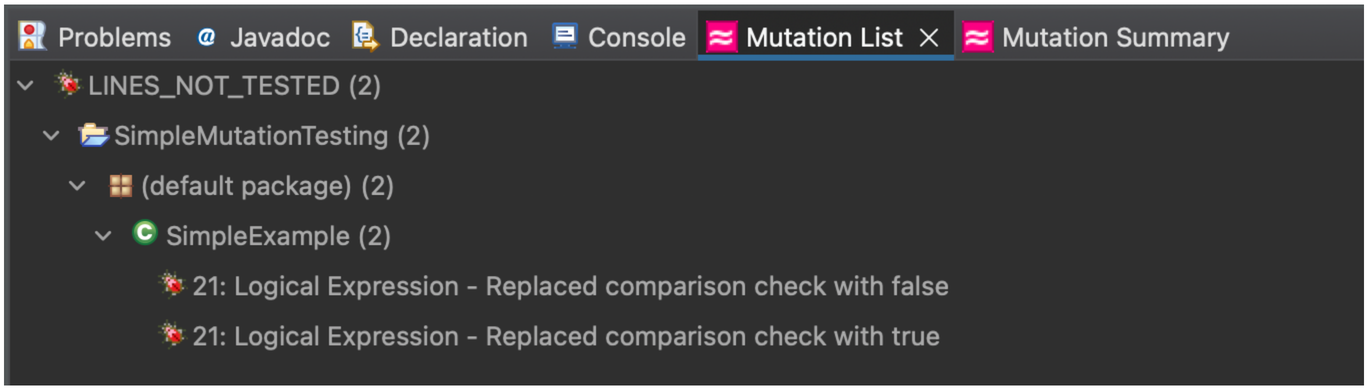
```
        public static boolean PositiveCheck(int number) {
                if (true) {                                     // --> true or false (2 cas
                        return true;                            // positive number
                }
                else {
                        return false;                           // zero or negative
                }
        }
```

Replacing the logical expression with FALSE:

```
        public static boolean PositiveCheck(int number) {
                if (false) {                                    // --> true or false (2 cas
                        return true;                            // positive number
                }
                else {
                        return false;                           // zero or negative
                }
        }
```

Without tests to execute this code, it will generate the following mutations in the LINES_NOT_TESTED group under the Mutations List tab.

The icons for unresolve mutants are shown as "red bugs".



```
LINES_NOT_TESTED (2)
  SimpleMutationTesting (2)
    (default package) (2)
      SimpleExample (2)
        21: Logical Expression - Replaced comparison check with false
        21: Logical Expression - Replaced comparison check with true
```

Figure 2.13.2: Example Code 2: Logical Expression Mutant (Remove Conditionals) without test cases

In order to fix the mutations, we can write the following test case assertions:

```
        // testEvenOddCheckWithEvenNumber tests for positive number
```

```
    // testEvenOddCheckWithEvenNumber tests for positive number
    @Test
    public void testPositiveCheckWithPositiveNumber() {
            assertTrue(SimpleExample.PositiveCheck(10));
    }

    // testEvenOddCheckWithOddNumber tests for zero
    @Test
    public void testPositiveCheckWithZero() {
            assertFalse(SimpleExample.PositiveCheck(0));
    }

    // testEvenOddCheckWithOddNumber tests for negative number
    @Test
    public void testPositiveCheckWithNegativeNumber() {
            assertFalse(SimpleExample.PositiveCheck(-5));
    }
```

If we run the mutation testing again then we will not get any of the previous mutations in the LINES_NOT_TESTED group under the Mutations List tab.

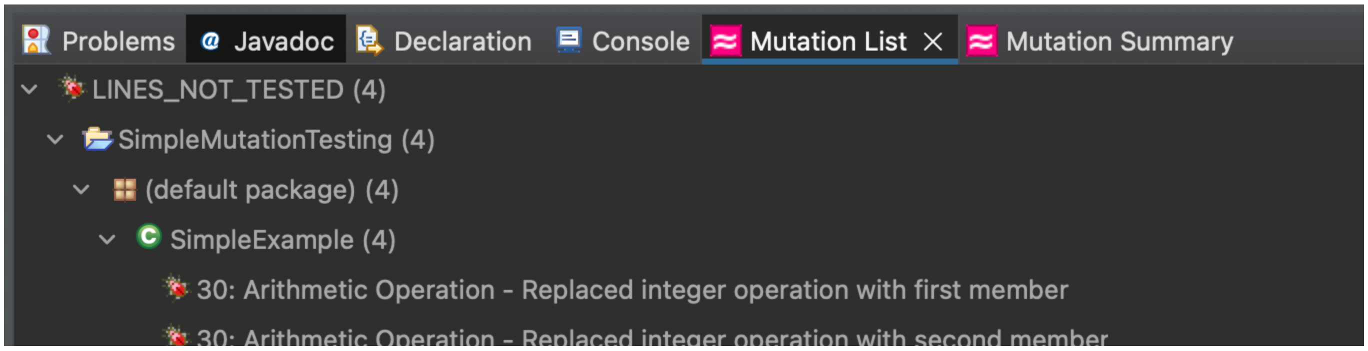
2.13.1.5. Example Code 3: Multiple Mutants in One (EvenOddCheck)

We can have programming statements where we have both arithmetic operation(s) and logical expression(s). In such cases, mutation testing will return mutants for each type and list them under the Mutations List tab.

For example, we want to write a function that takes a number and returns TRUE if the number is even and FALSE if the number is odd.

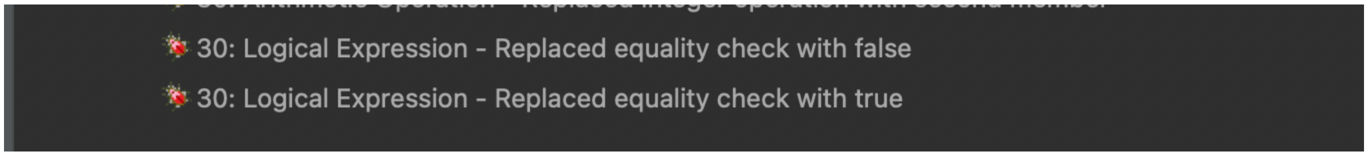
```
public static boolean EvenOddCheck(int number) {
        if (number % 2 == 0) {        // --> arithmetic operation (2 cases
                return true;                // even number
        }
        else {
                return false;               // odd number
        }
}
```

As a result, it will generate the following mutations under the Mutations List tab:



Problems | Javadoc | Declaration | Console | Mutation List | Mutation Summary

```
LINES_NOT_TESTED (4)
  SimpleMutationTesting (4)
    (default package) (4)
      SimpleExample (4)
        30: Arithmetic Operation - Replaced integer operation with first member
        30: Arithmetic Operation - Replaced integer operation with second member
```



30: Logical Expression - Replaced equality check with false
30: Logical Expression - Replaced equality check with true

Figure 2.13.3: Example Code 3: Multiple Mutants in One Statement without test cases

In order to fix the mutations, we can write the following test case assertions: