SENG 275

SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING

University of Victoria

MUTATION (FAULT-BASED)TESTING -**WHO WILL TEST THE TESTS THEMSELVES?**

University of Victoria

Code Coverage is not enough-The problem of zero assertions !!

- 100% Code Coverage does NOT mean high test suite quality!
- Setting coverage goals for a team will backfire if the team does not have the adequate mindset (and skillset) in test automation.

University of Victoria

Code Coverage is not enough-The problem of zero assertions !!

“A team with no prior knowledge in unit testing is "forced" into writing unit tests with 100% code coverage - because a manager heard somewhere that code coverage is good and that 100% code coverage equals quality. So, the team - who doesn't have the skillset in writing unit tests - simply went through the Code Coverage report and added code to execute every branch in every method in every class. So that team achieved 100% Code Coverage, because, yes, the code was getting executed. But there were zero assertions!”

-Some test observer

University of Victoria

Superficial test coverage-when managers say, “85% coverage is the shipping gate.”

- Test only **happy paths** - only expected behaviour tested.
- **Missing assertions**

```
public void TestFunction()
{
        //DoNothing
}
```

- Write **quick and simple** tests for easy to satisfy conditions. Leave the complex ones.
- **Mocking** misuse.
- Tests which have no business value- **low quality** tests.

Thus, test suite quality becomes inferior.

100% coverage

Mutation testing-checking the **fault-detection capability** of your test suite.

- Mutation testing is a **structural** testing method.
- Mutation (Fault) is a small **change** in the code.
- Mutations are automatically **seeded** into your code, then your tests are run.
- If your tests fail then the mutation is **killed**, if your tests pass then the mutation **lived**.
- The **quality** of your tests can be gauged from the **percentage** of mutations killed.
- Mutants should **guide** the tester towards an **effective test suite**.
- Mutation testing makes your code ready for any **future changes**.

University of Victoria

Steps

1. We take a program, and a test suite is generated for that program (using other test techniques) and we achieve 100%-line coverage using that test suite.
2. We (normally an automated tool) create a number of similar programs (mutants), each differing from the original in one small way, i.e., each possessing a fault.
   - E.g., replacing an addition operator by a multiplication operator.
3. The original test data are then run on the mutants.
4. If test cases detect differences in mutants, then the mutants are said to be dead (killed), and the test set is considered adequate otherwise more test cases are devised to kill the mutants.
5. In rare cases if adding more test cases do not kill all mutants, you may consider strengthening your source code.

University of Victoria

Mutant types

- Mutant is simply the mutated version of the source code.

Survived mutants	Killed mutants	Equivalent mutants
• Mutants that are still alive after running test data through the original and mutated variants of the source code. • These must be killed.	• These are mutants that are killed after mutation testing.	• They have the same meaning as the original source code, even though they may have different syntax. • These are stubborn and cannot be killed.

Equivalent mutants

```
int foo(int x) {
  return x * 1;
}
int foo_mutant(int x) {
  return x + 0;
}
```

```
int bar(int x) {
  return x + 1;
}
int bar_mutant(int x) {
  return x -(-1) ;
}
```

Types of Mutation Testing

- **Value Mutations −** In this testing, the values are modified to find errors in the program. A small value is modified to a larger value or vice-versa. Generally, **constants** are changed in value mutation testing.
- **Decision Mutations −** In this testing, **logical/arithmetic operators** are modified to discover errors in the program.
- **Statement Mutations −** In this testing, a statement is deleted or is replaced by another statement.

Example-Decision mutants

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

University of Victoria

Example-Decision mutants

Version	Code
P (original)	`int sum(int a, int b) { return a + b; }`
Mutant 1	`int sum(int a, int b) { return a - b; }`
Mutant 2	`int sum(int a, int b) { return a * b; }`
Mutant 3	`int sum(int a, int b) { return a / b; }`
Mutant 4	`int sum(int a, int b) { return a + b++; }`

	Test data (a,b)			
	(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
P	2	0	-1	-2
M1	0	0	-1	0
M2	1	0	0	1
M3	1	Error	Error	1
M4	2	0	-1	-2

- The major difficulties appear with the detection of functionally equivalent mutants

A program and four mutants

Version	Code
P (original)	int sum(int a, int b) { return a + b; }
Mutant 1	int sum(int a, int b) { return a - b; }
Mutant 2	int sum(int a, int b) { return a * b; }
Mutant 3	int sum(int a, int b) { return a / b; }
Mutant 4	int sum(int a, int b) { return a + b++; }

	Test data (a,b)			
	(1, 1)	(0, 0)	(-1, 0)	(-1, -1)
P	2	0	-1	-2
M1	0	0	-1	0
M2	1	0	0	1
M3	1	Error	Error	1
M4	2	0	-1	-2

Mutation coverage and score

- Given a mutant m of a derivation d, a test is said to kill the mutant if and only if this test produces a different output on m than on d.
- Mutation coverage requires every mutant to be killed by at least one test.
- Mutation Score = (Killed Mutants / Total number of Mutants) * 100
- If mutation score is 100%, then test cases are considered to be mutation adequate.

University of Victoria

Mutators active by default when running a PIT mutation coverage test

- *INCREMENTS_MUTATOR*
- *VOID_METHOD_CALL_MUTATOR*
- *RETURN_VALS_MUTATOR*
- *MATH_MUTATOR*
- *NEGATE_CONDITIONALS_MUTATOR*
- *INVERT_NEGS_MUTATOR*
- *CONDITIONALS_BOUNDARY_MUTATOR*

University of Victoria

Ref: https://pitest.org/quickstart/mutators/

Increments Mutator (INCREMENTS) example

```
public int method(int i) {
  i++;
  return i;
}
```

Will be mutated to

```
public int method(int i) {
  i--;
  return i;
}
```

Invert Negatives Mutator (INVERT_NEGS) example

```
public float negate(final float i) {
  return -i;
}
```

will be mutated to

```
public float negate(final float i) {
  return i;
}
```

Conditional boundary mutator-replaces the relational operators <, <=, >, >= with their boundary counterpart

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

Initial Code:

```
if(a < b)
 c = 10;
else
 c = 20;
```

Changed Code:

```
if(a > b)
 c = 10;
else
 c = 20;
```

University of Victoria

Conditional boundary example

- Original code (a>=10) Test case a=16 passes.
- Mutated code (a>10) Test case a=16 passes. Therefore, this test case is not able to detect this mutation. So, we must add another test case to kill this mutant.
- Add test case a=10. this tests boundary and kills the mutant.

University of Victoria

Math Mutator (MATH)

Original conditional	Mutated conditional
+	-
-	+
*	/
/	*
%	*
&	|
|	&
^	&
<<	>>
>>	<<
>>>	<<

University of Victoria

Negate Conditionals Mutator (NEGATE_CONDITIONALS)

Original conditional	Mutated conditional
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

These mutations are generally very easily detected by test suites.

Negated conditionals

If **a>5**
  return true
else
  return false

Is changed to

If **a<=5**
  return true
else
  return false

Test case a=2 returns false

Test case a=60 returns true

Test case a=2 returns true

Test case a=60 returns false

Mutation killed

University of Victoria

Some negated mutants might survive.
Code to check whether a string is a palindrome, test case inputString="noon"

```
public boolean isPalindrome(String inputString) {
    if (inputString.length()==0) {
        return true;
    }
else {
        char firstChar = inputString.charAt(0);
        char lastChar = inputString.charAt(inputString.length() - 1);
        String mid = inputString.substring(1, inputString.length() - 1);
        return (firstChar == lastChar) && isPalindrome(mid);
    }
}
```

Test case which is a palindrome

```
@Test
   public void whenPalindrom_thenAccept() {
      assertTrue(C.isPalindrome(“noon"));
   }
```

Mutated code through negated conditional. This will still return true even though the code has changed. Mutant survives. It is difficult to kill this mutant because it returns true for any non-zero length string whether it is a palindrome or not. This means there is a weakness in the source code itself.

```
public boolean isPalindrome(String inputString) {
    if (inputString.length!=0) {
        return true;
    }
else {
        char firstChar = inputString.charAt(0);
        char lastChar = inputString.charAt(inputString.length() - 1);
        String mid = inputString.substring(1, inputString.length() - 1);
        return (firstChar == lastChar) && isPalindrome(mid);
    }
}
```

University of Victoria

Void Method Call Mutator (VOID_METHOD_CALLS) example-removes method calls to void methods

```
public void someVoidMethod(int i) {
  // does something
}

public int foo() {
  int i = 5;
  someVoidMethod(i);
  return i;
}
```

```
public void someVoidMethod(int i) {
  // does something
}

public int foo() {
  int i = 5;
  return i;
}
```

Since a void method does not return anything, if the test is only for foo() in this case, you might miss testing someVoidMethod(i). This mutation makes sure you are testing a return void method too.

Return Values Mutator (RETURN_VALS)

Return Type	Mutation
boolean	replace the unmutated return value true with false and replace the unmutated return value false with true
Int, byte, short	if the unmutated return value is 0 return 1, otherwise mutate to return value 0
long	replace the unmutated return value x with the result of x+1
Object	replace non-null return values with null

University of Victoria

Example

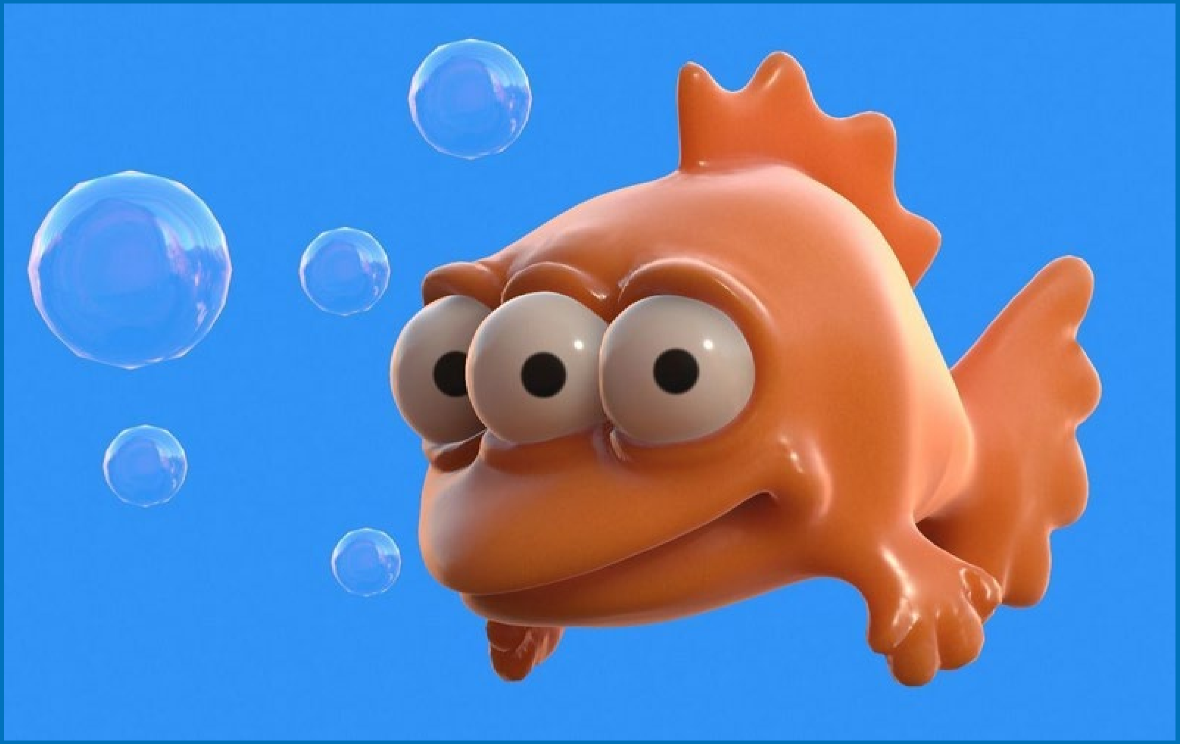
```
public Object foo() {
  return new Object();
}
```

will be mutated to

```
public Object foo() {
  new Object();
  return null;
}
```

University of Victoria

TEST CALCULATOR



University of Victoria

Calculator program- sum, multiply, subtract

```
public class Calculator {
public int doSum(int a, int b)
{
return a + b;
}
public int doSub(int a, int b)
{
return a - b ;
}
public int doProduct(int a, int b)
{
return a * b;
}
public Boolean compareTwoNums(int a, int b)
{
return a == b;
}
}
```

Calculator program- sum, multiply, subtract

```
public class CalculatorTest {
Calculator c=new Calculator();

@Test
public void testSum()
{
int expected=120;
int actual=c.doSum(30,40);
assertEquals(actual,expected);
}
@Test
public void testSub()
{
int expected=0;
int actual=c.doSub(0,0);
assertEquals(expected,actual);
System.out.println("The Sum is:
"+actual);
}
```

```
    @Test
    public void testProduct()
    {
    int expected=35;
    int actual=c.doProduct(5,7);
    assertEquals(actual,expected);
    }
    @Test
    public void testCompareTrue()
    {
    boolean actual=c.compareTwoNums(12,12);
    assertTrue(actual);
    }
```

Running PITest

- Run CalculatorTest with coverage and get 100%coverage.
- Run->Edit Configurations->PIT Runner->CalculatorPitest
- Run CalculatorPitest->Open report in browser

Run to determine whether we have 100% line coverage

Class CalculatorTest

all > default-package > CalculatorTest

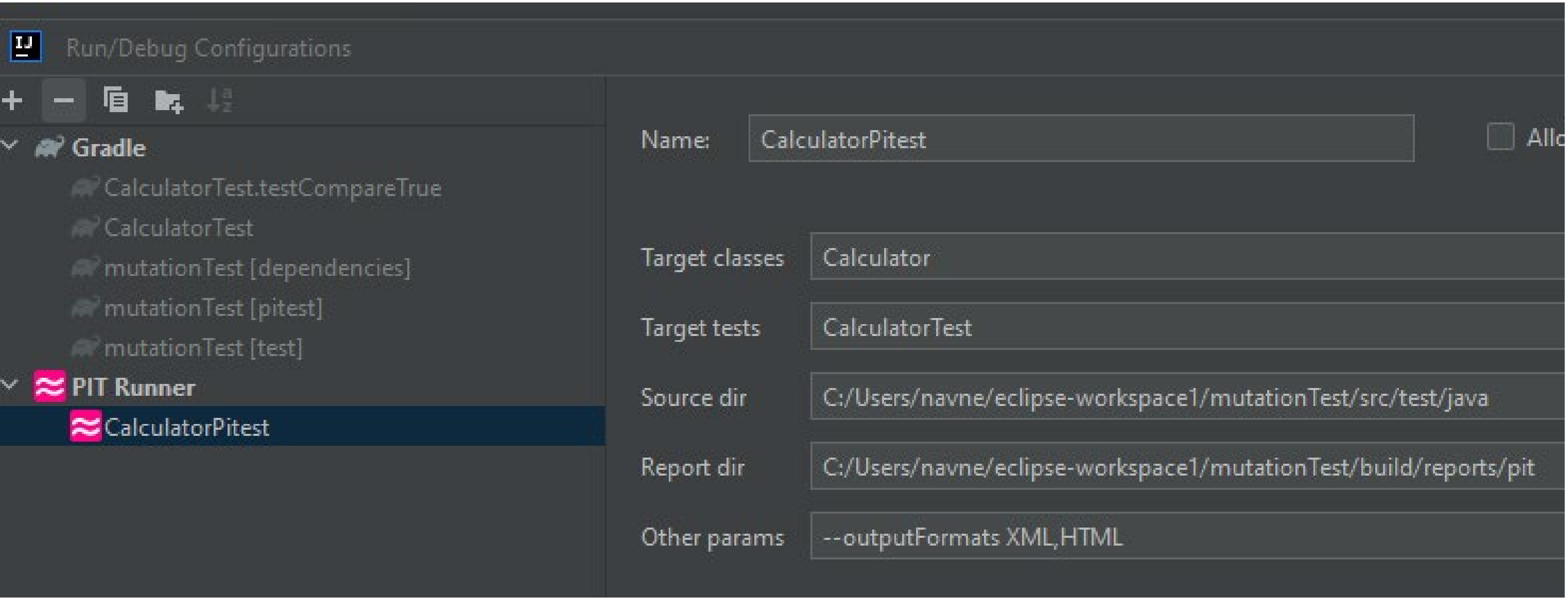
4	0	0	0.194s
tests	failures	ignored	duration

100%
successful

Tests | Standard output

Test	Duration	Result
testCompareTrue()	0.023s	passed
testProduct()	0.002s	passed
testSub()	0.166s	passed
testSum()	0.003s	passed

Gradle configuration



Run->CalculatorPiTest

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	5/5	63%	5/8	63%	5/8

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	5/5	63%	5/8	63%	5/8

Calculator.java

Mutations

4	1. replaced int return with 0 for Calculator::doSum → KILLED 2. Replaced integer addition with subtraction → KILLED
7	1. Replaced integer multiplication with division → KILLED 2. replaced int return with 0 for Calculator::doProduct → KILLED
9	1. Replaced integer subtraction with addition → SURVIVED 2. replaced int return with 0 for Calculator::doSub → SURVIVED
12	1. negated conditional → KILLED 2. replaced Boolean return with True for Calculator::compareTwoNums → SURVIVED

University of Victoria

Calculator program

```
public class Calculator {
public int doSum(int a, int b)
{
return a + b;
}
public int doSub(int a, int b)
{                                   (0,0)
return a - b ;
}
public int doProduct(int a, int b)
{
return a * b;
}
public Boolean compareTwoNums(int a, int b)
{
return a == b;        (12,12)
}
}
```

Calculator.java

Mutations

```
4   1. replaced int return with 0 for Calculator::doSum → KILLED
    2. Replaced integer addition with subtraction → KILLED
7   1. Replaced integer multiplication with division → KILLED
    2. replaced int return with 0 for Calculator::doProduct → KILLED
9   1. Replaced integer subtraction with addition → SURVIVED
    2. replaced int return with 0 for Calculator::doSub → SURVIVED
12  1. negated conditional → KILLED
    2. replaced Boolean return with True for Calculator::compareTwoNums → SURVIVED
```

Math Mutator

- The reason the mutant for addition is surviving is because *addition (x+y) operation* is different from the *subtraction (x-y)* operation but a test input of (0,0) gives the same result for both the cases.

Add another test for subtraction

```
@Test
    public void testSub2()
    {
        int expected=10;
        int actual=c.doSub(40,30);
        assertEquals(expected,actual);
        System.out.println("The Sum is: "+actual);
    }
```

University of Victoria

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	5/5	88%	7/8	88%	7/8

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	5/5	88%	7/8	88%	7/8

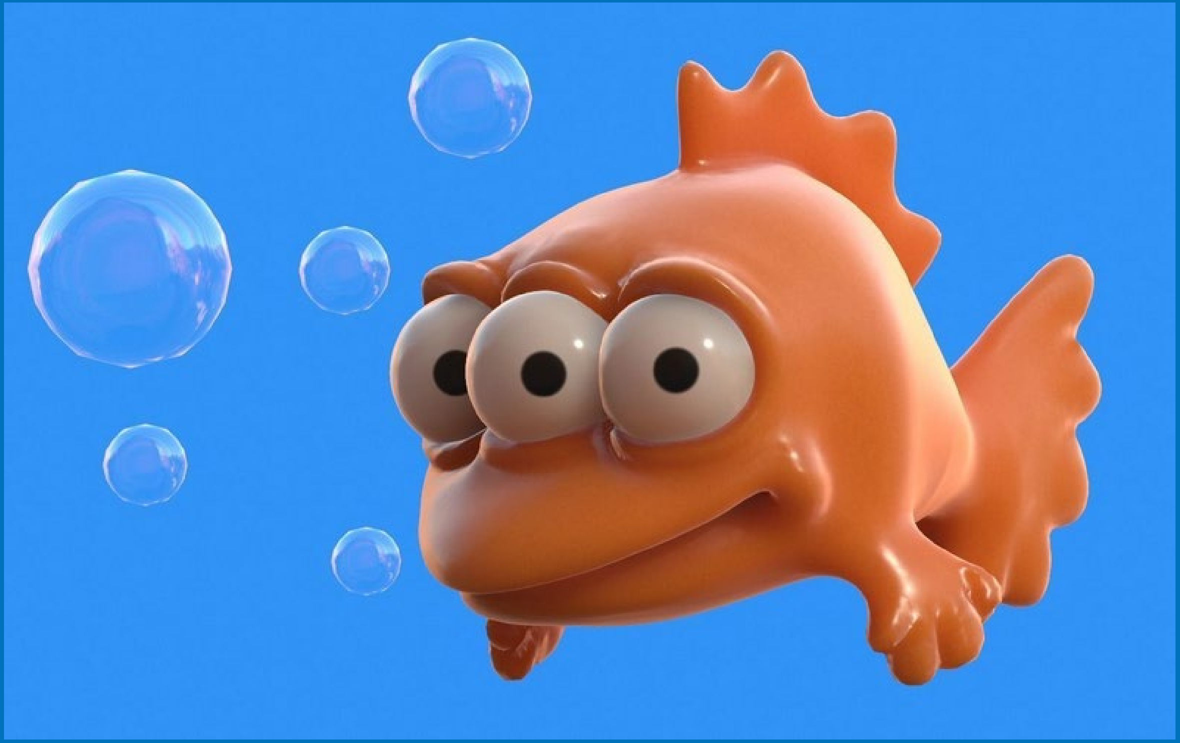
Calculator.java

Mutations

4	1. replaced int return with 0 for Calculator::doSum → KILLED 2. Replaced integer addition with subtraction → KILLED
7	1. Replaced integer multiplication with division → KILLED 2. replaced int return with 0 for Calculator::doProduct → KILLED
9	1. Replaced integer subtraction with addition → KILLED 2. replaced int return with 0 for Calculator::doSub → KILLED
12	1. negated conditional → KILLED 2. replaced Boolean return with True for Calculator::compareTwoNums → SURVIVED

University of Victoria

TEST COMPARE METHOD



Add another test for compare where two numbers are not equal

```
@Test
    public void testCompareTrue2()
    {
        boolean actual=C.compareTwoNums(12,1);
        assertFalse(actual);
        System.out.println("The Comparison is: "+actual);
    }
```

Pit Test Coverage Report

Package Summary

default

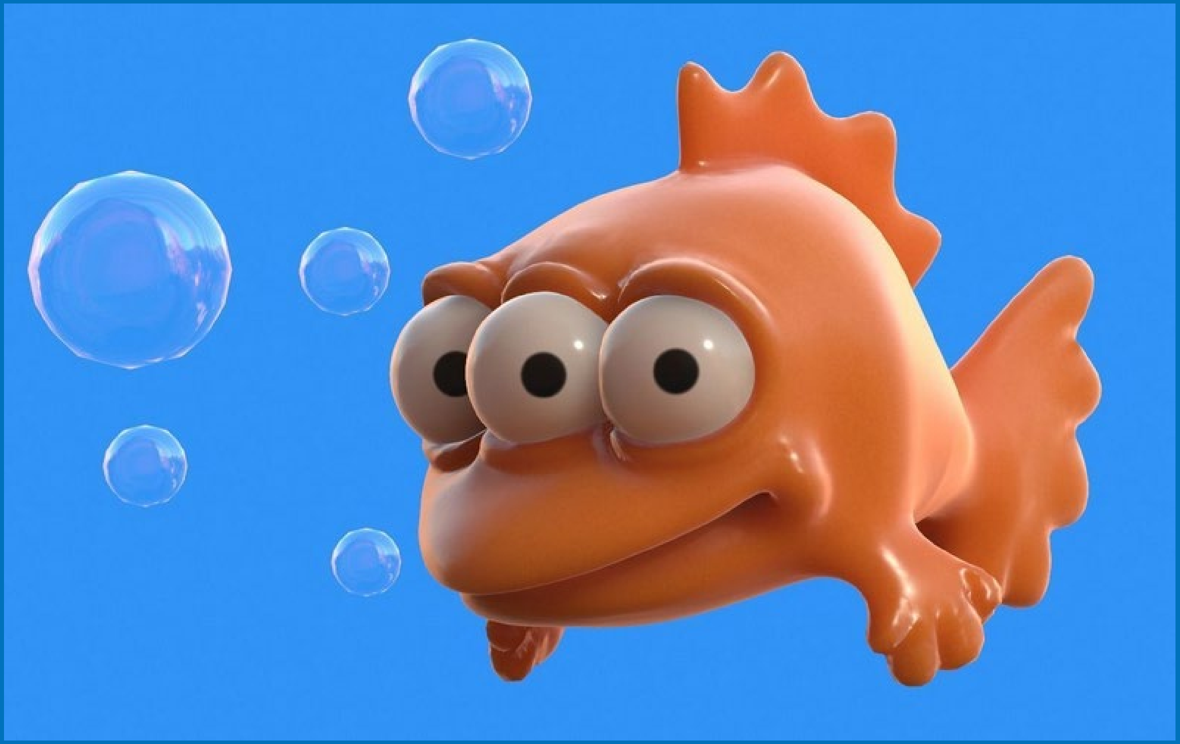
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	5/5	100%	8/8	100%	8/8

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	5/5	100%	8/8	100%	8/8

University of Victoria

REMOVE ASSERTION



Calculator program

```
public class CalculatorTest {
Calculator c=new Calculator();

@Test
public void testSum()
{
int expected=120;
int actual=c.doSum(30,40);
assertEquals(actual,expected);
}
```

Removed

```
@Test
public void testSub()
{
int expected=0;
int actual=c.doSub(0,0);
assertEquals(expected,actual);
System.out.println("The Sum is: "+actual);
}
```

```
@Test
public void testProduct()
{
int expected=35;
int actual=c.doProduct(5,7);
assertEquals(actual,expected);
}
@Test
public void testCompareTrue()
{
boolean actual=c.compareTwoNums(12,12);
assertTrue(actual);
}
```

We get 100% code coverage even after removing assertion

Class CalculatorTest

all > default-package > CalculatorTest

6	0	0	0.125s
tests	failures	ignored	duration

100%
successful

Tests | Standard output

Test	Duration	Result
testCompareTrue1()	0.006s	passed
testCompareTrue2()	0.017s	passed
testProduct()	0.002s	passed
testSub1()	0.043s	passed
testSub2()	0.003s	passed
testSum()	0.054s	passed

Do mutation testing- we are not able to fool mutation testing

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	5/5	75%	6/8	75%	6/8

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	5/5	75%	6/8	75%	6/8

University of Victoria

Calculator.java

Mutations

4	1. replaced int return with 0 for Calculator::doSum → SURVIVED 2. Replaced integer addition with subtraction → SURVIVED
7	1. Replaced integer multiplication with division → KILLED 2. replaced int return with 0 for Calculator::doProduct → KILLED
9	1. Replaced integer subtraction with addition → KILLED 2. replaced int return with 0 for Calculator::doSub → KILLED
12	1. negated conditional → KILLED 2. replaced Boolean return with True for Calculator::compareTwoNums → KILLED

Active mutators

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

Calculator program- sum, multiply, subtract

```
public class Calculator {
public int doSum(int a, int b)
{
return a + b;
}
public int doSub(int a, int b)
{
return a - b ;
}
public int doProduct(int a, int b)
{
return a * b;
}
public Boolean compareTwoNums(int a, int b)
{
return a == b;
}
}
```

No assertion for killing these mutations.

Mutations

```
4   1. replaced int return with 0 for Calculator::doSum → SURVIVED
    2. Replaced integer addition with subtraction → SURVIVED
7   1. Replaced integer multiplication with division → KILLED
    2. replaced int return with 0 for Calculator::doProduct → KILLED
9   1. Replaced integer subtraction with addition → KILLED
    2. replaced int return with 0 for Calculator::doSub → KILLED
12  1. negated conditional → KILLED
    2. replaced Boolean return with True for Calculator::compareTwoNums → KILLED
```

University of Victoria

Calculator program

```java
public class CalculatorTest {
Calculator c=new Calculator();

@Test
public void testSum()
{
int expected=120;
int actual=c.doSum(30,40);
assertEquals(actual,expected);
}
@Test
public void testSub()
{
int expected=0;
int actual=c.doSub(0,0);
assertEquals(expected,actual);
System.out.println("The Sum is:
"+actual);
}
```

Add back

```java
    @Test
    public void testProduct()
    {
    int expected=35;
    int actual=c.doProduct(5,7);
    assertEquals(actual,expected);
    }
    @Test
    public void testCompareTrue()
    {
    boolean actual=c.compareTwoNums(12,12);
    assertTrue(actual);
    }
```

When we add assertions back and run mutation test again

Pit Test Coverage Report

Package Summary

default

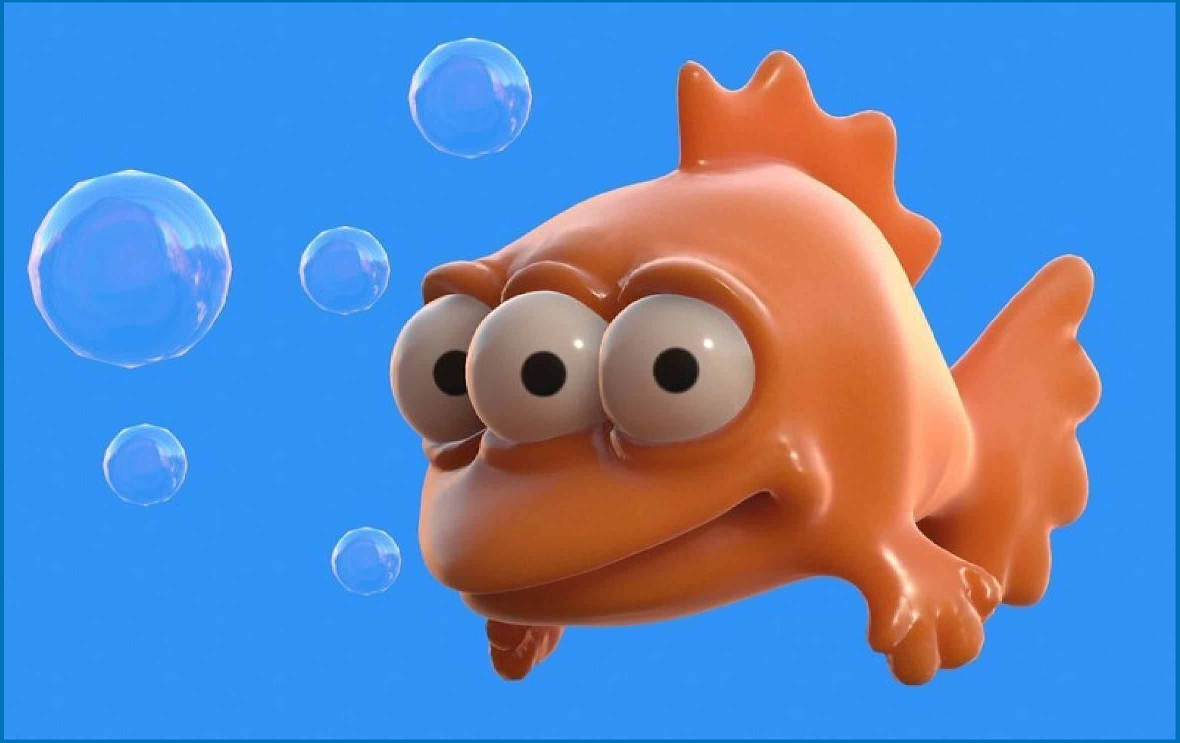
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	5/5	100%	8/8	100%	8/8

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	5/5	100%	8/8	100%	8/8

University of Victoria

OTHER MUTATIONS



University of Victoria

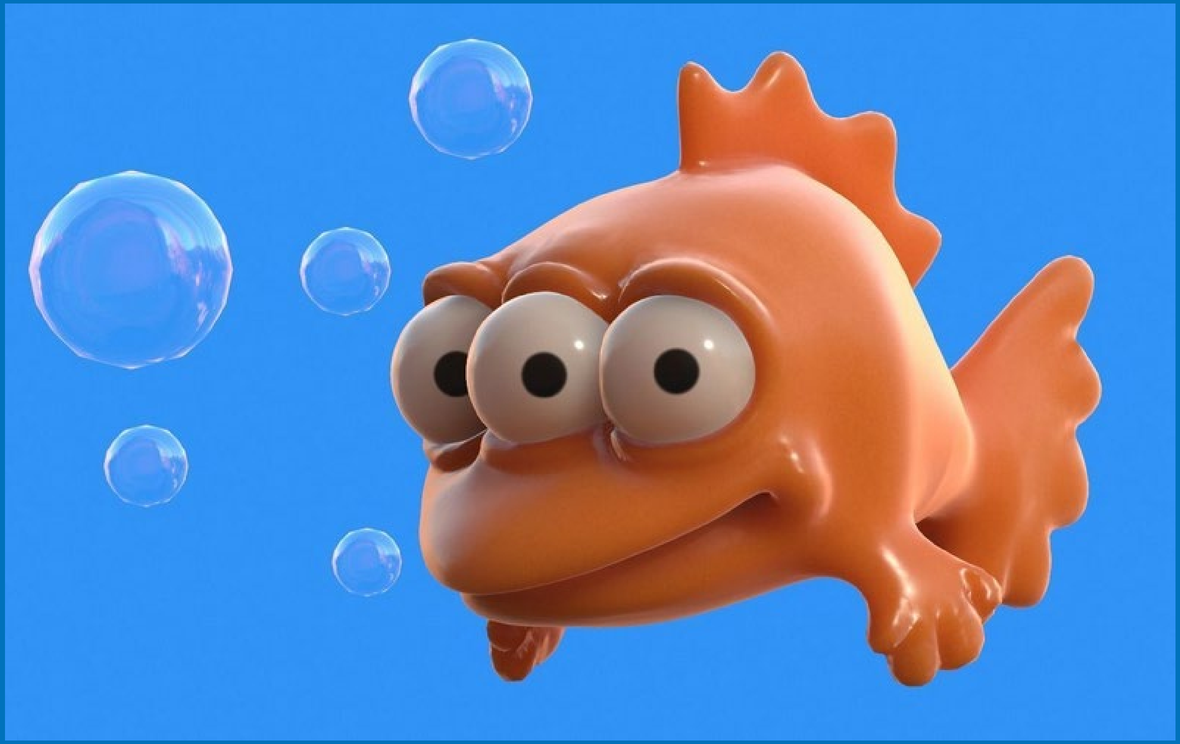
Replace multiplication operator with exponential operator- Which test case will survive this mutation? (2,2) or (1,1). So, add a test case (4,3) to kill this mutant.

```
public int doProduct(int a, int b)
{
return a * b;
}
```

Change to

```
public int doProduct(int a, int b)
{
return Math.pow(a,b);
}
```

TEST FOR BOUNDARIES



University of Victoria

Conditionals Boundary Mutator- add a method which checks whether a number is natural

```
 //Finding if a number is a natural number.
// Natural numbers are more than or equal to 0

public Boolean isNatural(int a) {
        Boolean result=false;
        if(a>=0)
        {
           result=true;
        }
```

University of Victoria

Add a test case for it

```
@Test
    public void testNatural()
    {
        boolean actual=c.isNatural(12);
        assertTrue(actual);
        System.out.println("The result is: "+actual);
    }
```

University of Victoria

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	9/9	91%	10/11	91%	10/11

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	9/9	91%	10/11	91%	10/11

Calculator.java

Mutations

4	1. replaced int return with 0 for Calculator::doSum → KILLED 2. Replaced integer addition with subtraction → KILLED
7	1. Replaced integer multiplication with division → KILLED 2. replaced int return with 0 for Calculator::doProduct → KILLED
9	1. Replaced integer subtraction with addition → KILLED 2. replaced int return with 0 for Calculator::doSub → KILLED
12	1. negated conditional → KILLED 2. replaced Boolean return with True for Calculator::compareTwoNums → KILLED
18	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED
22	1. replaced Boolean return with False for Calculator::isNatural → KILLED

University of Victoria

Add boundary test

```
@Test
   public void testNatural2()
   {
      boolean actual=c.isNatural(0);
      assertTrue(actual);
      System.out.println("The result is: "+actual);
   }
```

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	9/9	100%	11/11	100%	11/11

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Calculator.java	100%	9/9	100%	11/11	100%	11/11

University of Victoria

Disadvantages of Mutation testing:

1. **Complex** mutations are difficult to implement. Some mutants **may not be killed**.
2. Mutation testing is **time-consuming and expensive**.
3. As this method includes the source code changes, it **can't** be used for the **black box testing.**
4. Being complicated to perform, this type of testing must be **automated**.

University of Victoria

Cost of Mutation Testing

Let's assume we have:

- a code base with 300 Java classes
- 10 test cases for each class
- on average, each test case requires 0.2 seconds for its execution
- the **total test suite execution costs** 300 * 10 * 0.2 = 600 seconds (**10 minutes**)

Let's assume we have, on average, 20 mutants per each class.

The **total cost of mutation analysis** is

300 * 10 * 0.2 * 20 = 12000 seconds (**3h 20 min**)

University of Victoria

Ref: Birgit Kratz