Syntax-based test coverage program mutation testing

- Basics: concepts of mutation, mutant, mutant kill.
- Mutation operators: examples, expressiveness
- Tool for mutation testing: PIT
- Subsumption of other coverage criteria using program-based mutation
- Mutation operators for integration testing

Ciências ULisboa

slides: Eduardo Marques, Vasco Vasconcelos, Francisco Martins, João Neto

Syntax-based coverage

- Approach
  - Test requirements are derived from the syntax of software artefacts.
- Mutation testing
  - Considers the mutation of source code, seeking to induce faults.
  - A mutant is the result of applying a single mutation.
  - A mutant is killed if there is a test case for which the test results are different from the original code.
  - Basic idea: if a significant share of mutants are not killed by a test set, the test set is also likely to be insensitive to real faults.
- Grammar-based testing
  - Inputs for the SUT are defined by a grammar.
  - Testing seeks to exercise productions of the grammar.
  - Grammar-based mutations are also considered to test the SUT with invalid inputs.

Program mutation - introduction

```
public static int numZeros(int[] x) {
   int count = 0;
   for (int i = 0; i < x.length; i++)
      if (x[i] == 0)
         count++;
   return count;
}
```

a "fault" is introduced by mutating the code.

```
public static int numZeros(int[] x) {
   int count = 0;
   for (int i = 1; i < x.length; i++)
      if (x[i] == 0)
         count++;
   return count;
}
```

Program mutation - introduction

```
public static int numZeros(int[] x) {
   int count = 0;
   for (int i = 1; i < x.length; i++)
     if (x[i] == 0)
       count++;
   return count;
 }
```

- **i=1** is a **mutation** of **i=0**; the code obtained by changing **i=0** to **i=1** (applying a mutation) is called a **mutant** of `numZeros()`.
- We say a test **kills the mutant** if the mutant yields outputs different from those of the original code.
  - Consider **`x={1,0,0}`**. The mutant is **not killed**; both the original code and the mutant return **2**.
  - Consider **`x={0,1,0}`**. The mutant is **killed**; the result is **1** rather than **2**.

original code

```
public static
int min(int x, int y) {
   int v;
   if (x < y)
     v = x;
   else
     v = y;
   return v;
}
```

mutants

m1

```
public static
int min(int x, int y) {
   int v;
   if (x >= y)
     v = x;
   else
     v = y;
   return v;
}
```

m2

```
public static
int min(int x, int y) {
   int v;
   if (x <= y)
     v = x;
   else
     v = y;
   return v;
}
```

m3

```
public static
int min(int x, int y) {
   int v;
   if (x < y)
     v = x;
   else
     v = -x;
   return v;
}
```

Example 2

Which mutants will be killed by tests:
(t1) (x,y) = (0,0)
(t2) (x,y) = (0,1)
(t3) (x,y) = (1,0)

Observe that **m2** can not be killed. Why not?

	x	y	min	m1	m2	m3
t1	0	0	0	0	0	0
t2	0	1	0	**1**	0	0
t3	1	0	0	**1**	0	**-1**

t1 kills none of the mutants.
t2 kills m1.
t3 kills m1 and m3.

Observe that m2 will always yield the same result as the original code. Thus it cannot be killed. It is a **functionally equivalent mutant**.

mutants

m1

```
public static
int min(int x, int y) {
  int v;
  if (x >= y)
    v = x;
  else
    v = y;
  return v;
}
```

m2

```
public static
int min(int x, int y) {
  int v;
  if (x <= y)
    v = x;
  else
    v = y;
  return v;
}
```

m3

```
public static
int min(int x, int y) {
  int v;
  if (x < y)
    v = x;
  else
    v = -x;
  return v;
}
```

Another example

```
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -
numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

✅
```
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

✅
```
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

✅
```
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

The test suite contains two tests for corner cases that throw an exception (i.e., `testInvertZero` and `testInvertMinValue`, and two “happy path” tests (i.e., `testInvert` and `testInvertNegative`). We will now determine the quality of our test suite using mutation testing.

```
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -
numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

First, create a *mutant* by applying a syntactic change to the original method. We want the syntactic change to be small: one operation/variable should be enough. Moreover, the syntactic change is a *change* that mimics mistakes that could be made by a programmer.

```
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator,
numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

First, create a *mutant* by applying a syntactic change to the original method. We want the syntactic change to be small: one operation/variable should be enough. Moreover, the syntactic change is a *change* that mimics mistakes that could be made by a programmer.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator,
numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

✅
```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

❌
```java
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```java
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

If we execute the test suite on this mutant, the `testInvertNegative` test will fail, as `result.getFloat()` would be positive instead of negative.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 0) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -
numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

Another mistake could be made in line 1. It is important to test the boundaries due to off-by-one errors. We can make a syntactic change by introducing such an off-by-one error. Instead of `numerator == 0`, in our new mutant we make it `numerator == 1`.

```java
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
1   if (numerator == 1) {
2     throw new ArithmeticException("...");
    }

3   if (numerator == Integer.MIN_VALUE) {
4     throw new ArithmeticException("...");
    }

5   if (numerator < 0) {
6     return new Fraction(-denominator, -
numerator);
    }

7   return new Fraction(denominator, numerator);
  }
}
```

```java
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}

@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}

@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}

@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

Another mistake could be made in line 1. It is important to test the boundaries due to off-by-one errors. We can make a syntactic change by introducing such an off-by-one error. Instead of `numerator == 0`, in our new mutant we make it `numerator == 1`.

```
public class Fraction {

  private int numerator;
  private int denominator;

  // ...

  public Fraction invert() {
    if (numerator == 1) {
      throw new ArithmeticException("...");
    }

    if (numerator == Integer.MIN_VALUE) {
      throw new ArithmeticException("...");
    }

    if (numerator < 0) {
      return new Fraction(-denominator, -
numerator);
    }

    return new Fraction(denominator, numerator);
  }
}
```

❌
```
@Test
public void testInvert(){
  Fraction f = new Fraction(1, 2);
  Fraction result = f.invert();
  assertEquals(2, result.getFloat(), 0.001);
}
```

✅
```
@Test
public void testInvertNegative(){
  Fraction f = new Fraction(-1, 2);
  Fraction result = f.invert();
  assertEquals(-2, result.getFloat(), 0.001);
}
```

❌
```
@Test(expected = ArithmeticException.class)
public void testInvertZero(){
  Fraction f = new Fraction(0, 2);
  f.invert();
}
```

✅
```
@Test(expected = ArithmeticException.class)
public void testInvertMinValue(){
  int n = Integer.MIN_VALUE;
  Fraction f = new Fraction(n, 2);
  f.invert();
}
```

We see that, again, the test suite catches this error. The test `testInvertZero` will fail, as it expects an exception but none is thrown in the mutant. The test `testInvert` will also fail since it has a 1 in the numerator which wrongly triggers an exception.

Exercise 1

```
public static int indexOf(int[] array, int v) {
  // (m1) i < array.length ---> i >= array.length
  // (m2) array[i] == v    ---> array[i] != v
  // (m3) i++              ---> i--
  for (int i = 0; i < array.length; i++)
    if (array[i] == v)
      return i;
  return -1;
}
```

For each mutant **`m1`**, **`m2`**, **`m3`**, if possible identify a test that:

1) kills the mutant (reaches the mutation, infects the state, leads to failure)
2) does not reach the mutation
3) reaches the mutation but does not lead to an error state
4) leads to an error state but not to failure (this is called a **weak kill**)

```
// (m1) i < array.length ---> i >= array.length
// (m2) array[i] == v     ---> array[i] != v
// (m3) i++               ---> i--
```

NPE: NullPointerException **AIOOBE**: ArrayIndexOutOfBoundsException

	array	v	expected	m1	m2	m3
t0	null	0	NPE	NPE (reach; no error)	NPE (not reach)	NPE (not reach)
t2	{}	0	-1	**AIOOBE** (kills)	-1 (not reach)	-1 (not reach)
t3	{1}	0	-1	-1 (error, no failure → weak kill)	**0** (kills)	**AIOOBE** (kills)
t4	{0}	0	0	**-1** (kills)	**-1** (kills)	0 (not reach)

- **m1:** always reached; **m2** and **m3** are only reached if **array != null && array.length > 0**
- **m1** is reached for **t0** but without infection and by **t3** with error (loop body is not executed) but not leading to failure (weak kill).
- If **m2** is reached, it always leads to a failure (causes an invalid return value necessarily).
- If **m3** is reached, it always leads to a failure (**array[-1]** is accessed causing an **AIOOBE**).

FACULDADE DE CIÊNCIAS UNIVERSIDADE DE LISBOA

Motivation for program-based mutation

- **Fundamental premise of mutation testing** [A&O, page 181]**:**

  "If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault" [provided we consider a rich set of mutation operators]

sensitivity to mutations (killing mutants)

$$\simeq$$

sensitivity to faults (exposing failures)

Motivation for program-based mutation

- Suppose you have a test set T for program P (maybe derived by applying some coverage criteria C, manually or automatically).
- Program-based mutation testing helps answering the following key question:
  - ***How "good" is T (and C)?***
- For each mutant $m \in M$ (M is the set of all mutants), if T is "good" then a test in T should kill m.
- If no test in T kills a mutant m, then T should be reformulated (one may also question the choice of criteria C)
- Program-based mutation is many times taken as the "golden standard" of coverage criteria.

Mutation coverage criteria

- **Mutation operator o**: takes program **P** yielding a set of mutants of **P**, **o(P)**
- Let **O** be the set of mutation operators and **M** be the set of all mutants generated using **O**, i.e., $\mathbf{M} = \{\mathbf{m} \mid \mathbf{m} \in \mathbf{o(P)}, \mathbf{o} \in \mathbf{O}\}$
- **Killing mutants**
  - A test **t** **kills** **m** if the output of **m** on **t** *differs* from the output of **P** on **t**
- **Mutation score** = percentage of mutants killed by a test set
- **Mutant coverage (MC)** criteria
  - For each mutant $\mathbf{m} \in \mathbf{M}$, the set TR includes one requirement that kills **m** (mutation score=100%)

Mutation operators and effectiveness

- **Mutants to avoid …**
  - **syntactically invalid**: would be caught by a compiler
  - **functionally-equivalent mutant:** no test can kill it
  - **trivial mutant:** almost every test can kill it
- **For effectiveness, a mutation operator should:**
  - **never** generate **syntactically invalid** mutants
  - generate **functionally-equivalent and trivial mutants with low probability**
  - **mimic typical programmer mistakes**
  - **not be subsumed by another operator**, i.e., tests that kill mutants created by the other operator also kill the ones generated by this one (or a large fraction of them)

The Costs of Mutation Testing

Mutation testing is not without its costs. We have to generate the mutants, possibly remove the functionally-equivalent mutants, and execute the tests against each mutant. This makes mutation testing quite expensive, i.e., it takes a long time to perform.

Assume we want to do some mutation testing. We have:
- A code base with 300 Java classes
- 10 test cases for each class
- Each test case takes 0.2 seconds on average

The total test execution time is then:
  300 x 10 x 0.2 = 600 seconds (10 minutes)
This execution time is for just the normal code.

The Costs of Mutation Testing

Mutation testing is not without its costs. We have to generate the mutants, possibly remove the functionally-equivalent mutants, and execute the tests against each mutant. This makes mutation testing quite expensive, i.e., it takes a long time to perform.

Assume we want to do some mutation testing. We have:
- A code base with 300 Java classes
- 10 test cases for each class
- Each test case takes 0.2 seconds on average

The total test execution time is then:

  300 x 10 x 0.2 = 600 seconds (10 minutes)

This execution time is for just the normal code.

For the mutation testing, we decide to generate on average 20 mutants per class. We will have to execute the entire test suite of a class on each of the mutants. Per class, we need

  20 x 10 x 0.2 = 40 seconds.

In total, the mutation testing will take

  300 x 40 = 12,000 seconds, or 3 hours and 20 minutes.

The Costs of Mutation Testing

Because of this cost, researchers have tried to find ways to make mutation testing faster

- A test case can never kill a mutant if it does not cover the changed statement. Run only tests that cover the changed statement.
- Once a test case kills a mutant, no need to run other tests.
- Mutants generated by the same operator and inserted at the same location are likely to be coupled with the same type of fault. So, when we use a certain mutation operator and replace the same statement with this mutant operator, we get two mutants that represent the same fault in the code.
  - It is then highly probable that if the test suite kills one of the mutants, it will also kill the others.
  - A heuristic that follows from this observation is to run the test suite against a subset of all the mutants (a technique also known as do fewer).
  - The simplest way of selecting the subset of mutants is by means of random sampling. As the name suggests, we select random mutants to consider. This is a very simple, yet effective way to reduce the execution time.

Mutation testing tools - basics

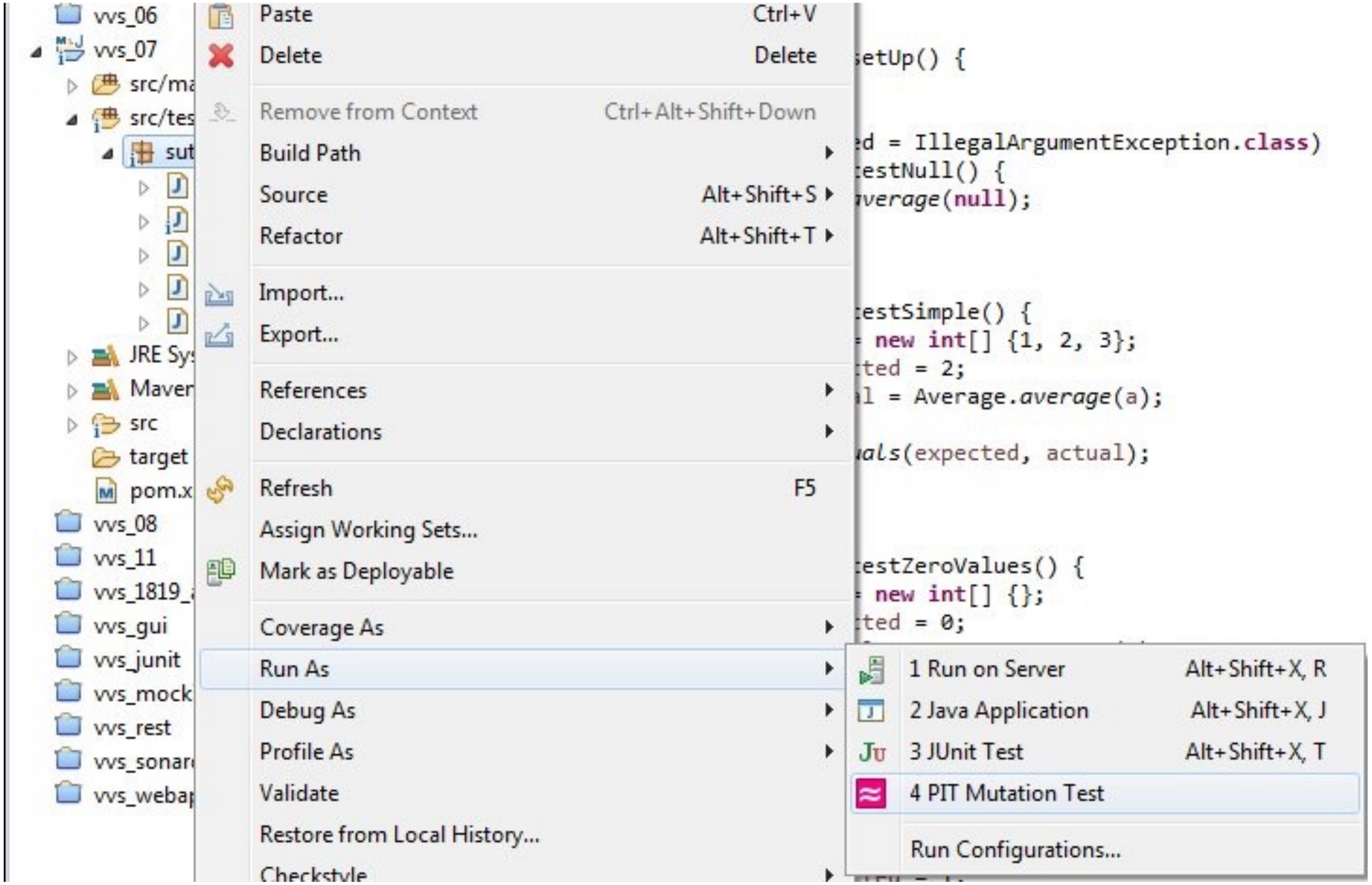
- A MT tool has a built-in set of mutation operators.
- The set of mutants for the SUT is generated in automated manner according to the mutation operators.
- A test set is run against the mutants. As soon as a mutant from the set is killed, it is typically not exercised by further tests.
- If the mutation coverage is not satisfactory, the test set is typically revised and/or increased with further test cases.
  - The strategies for both mutant generation and test selection/execution can be quite elaborate in technical terms.
- Let us have a look at PIT, a mutation testing tool:
  - `http://pitest.org/`

PIT mutation testing tool

- Homepage: http://pitest.org
- Eclipse plugin: https://github.com/philglover/pitclipse
- Main features:
  - Mutations over JVM bytecode
  - HTML reports of mutation coverage are generated with cross-reference to source code
  - well-picked “minimalistic” operators that do not generate too many mutants
  - fast ...
- Browse the PIT website and use the Eclipse plugin

PIT mutation testing tool

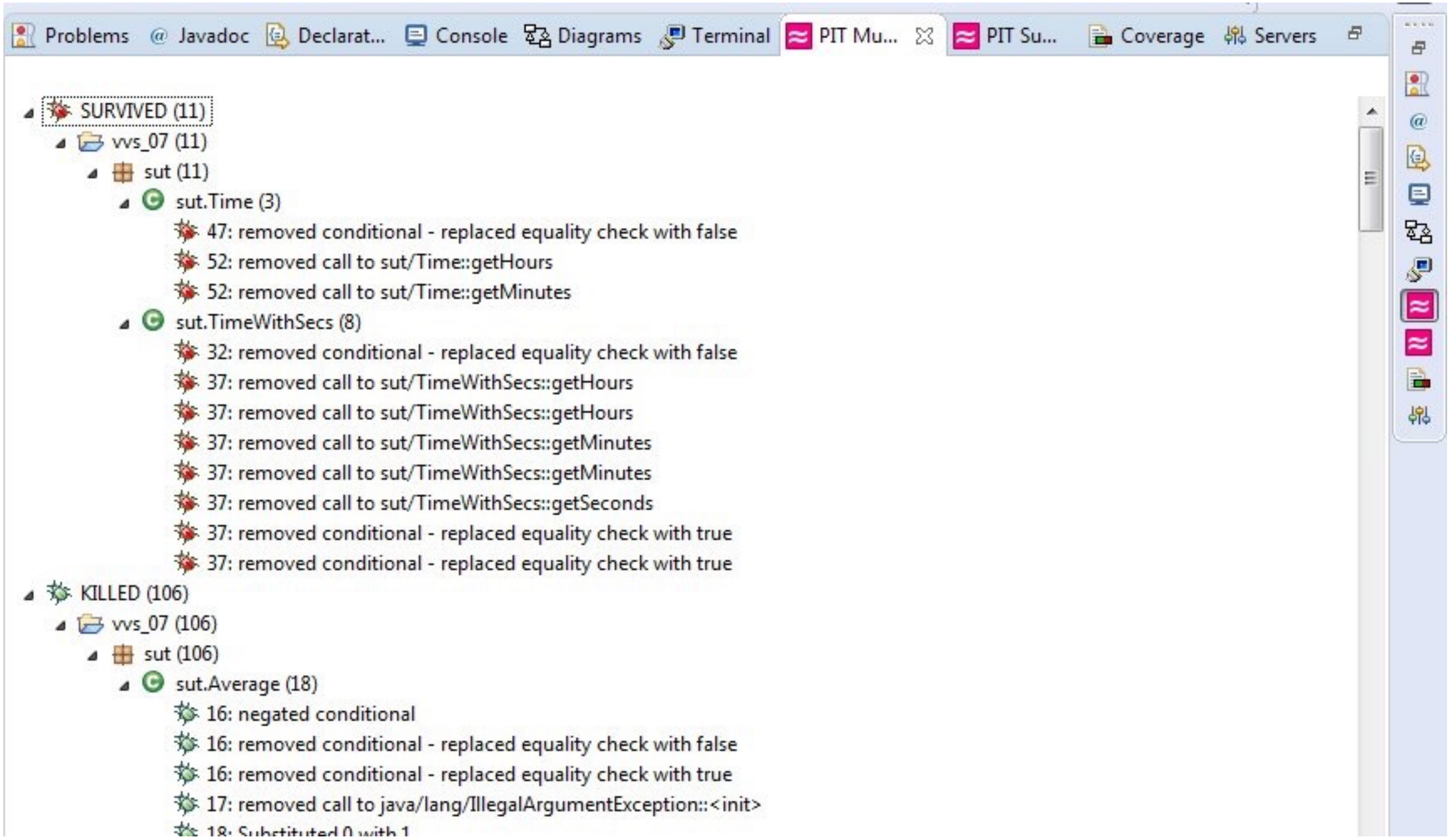
- Once installed:



```
etUp() {

ed = IllegalArgumentException.class)
estNull() {
verage(null);


estSimple() {
 new int[] {1, 2, 3};
ted = 2;
l = Average.average(a);

als(expected, actual);


estZeroValues() {
 new int[] {};
ted = 0;
```

PIT mutation testing tool

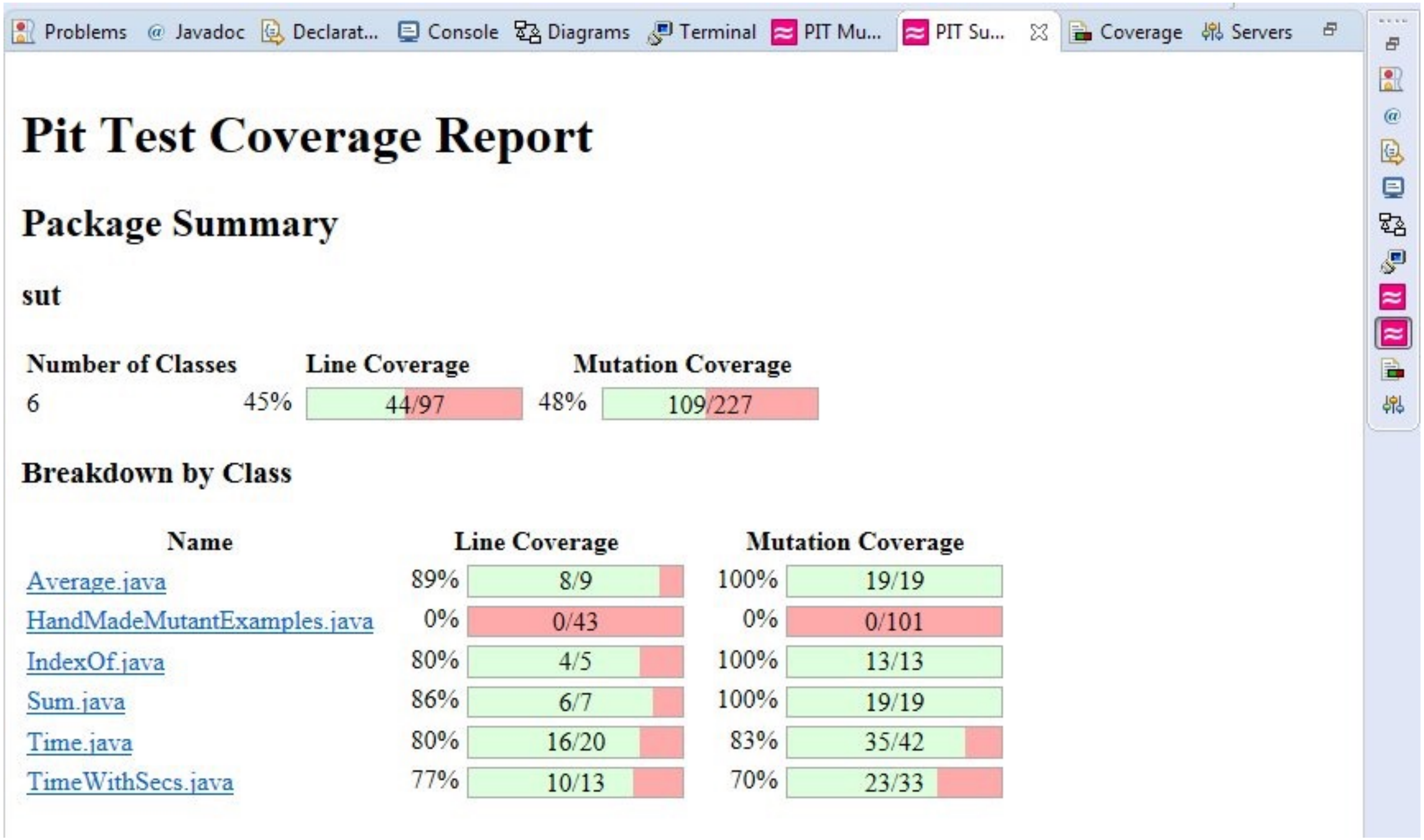
- The plugin reports which mutants were killed and which survived



```
SURVIVED (11)
  vvs_07 (11)
    sut (11)
      sut.Time (3)
        47: removed conditional - replaced equality check with false
        52: removed call to sut/Time::getHours
        52: removed call to sut/Time::getMinutes
      sut.TimeWithSecs (8)
        32: removed conditional - replaced equality check with false
        37: removed call to sut/TimeWithSecs::getHours
        37: removed call to sut/TimeWithSecs::getHours
        37: removed call to sut/TimeWithSecs::getMinutes
        37: removed call to sut/TimeWithSecs::getMinutes
        37: removed call to sut/TimeWithSecs::getSeconds
        37: removed conditional - replaced equality check with true
        37: removed conditional - replaced equality check with true
KILLED (106)
  vvs_07 (106)
    sut (106)
      sut.Average (18)
        16: negated conditional
        16: removed conditional - replaced equality check with false
        16: removed conditional - replaced equality check with true
        17: removed call to java/lang/IllegalArgumentException::<init>
        18: Substituted 0 with 1
```

PIT mutation testing tool

And an overall report with line and mutation coverage for each tested class



Pit Test Coverage Report

Package Summary

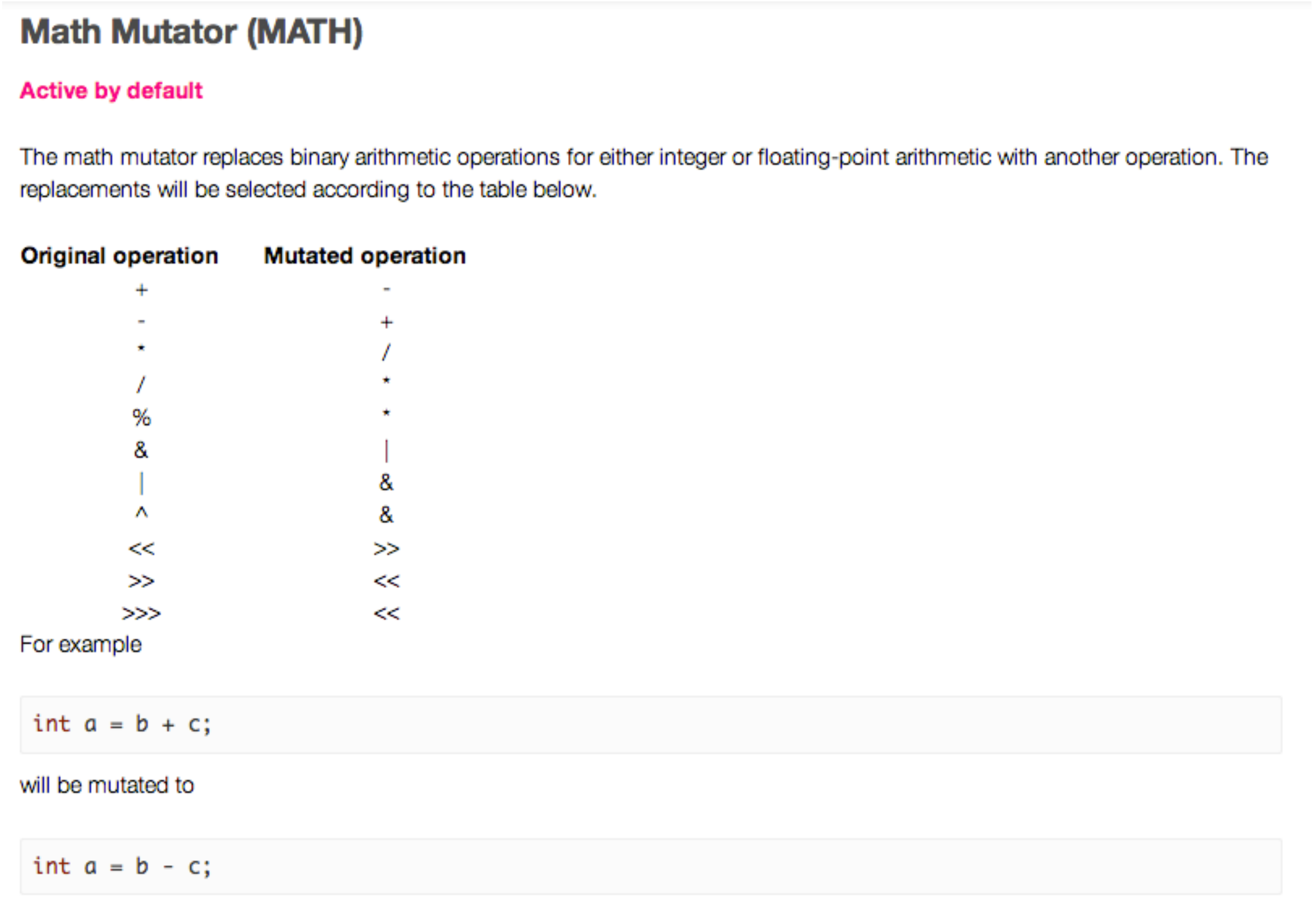
sut

Number of Classes	Line Coverage		Mutation Coverage	
6	45%	44/97	48%	109/227

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Average.java	89%	8/9	100%	19/19
HandMadeMutantExamples.java	0%	0/43	0%	0/101
IndexOf.java	80%	4/5	100%	13/13
Sum.java	86%	6/7	100%	19/19
Time.java	80%	16/20	83%	35/42
TimeWithSecs.java	77%	10/13	70%	23/33

PIT operators



Math Mutator (MATH)

Active by default

The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The replacements will be selected according to the table below.

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

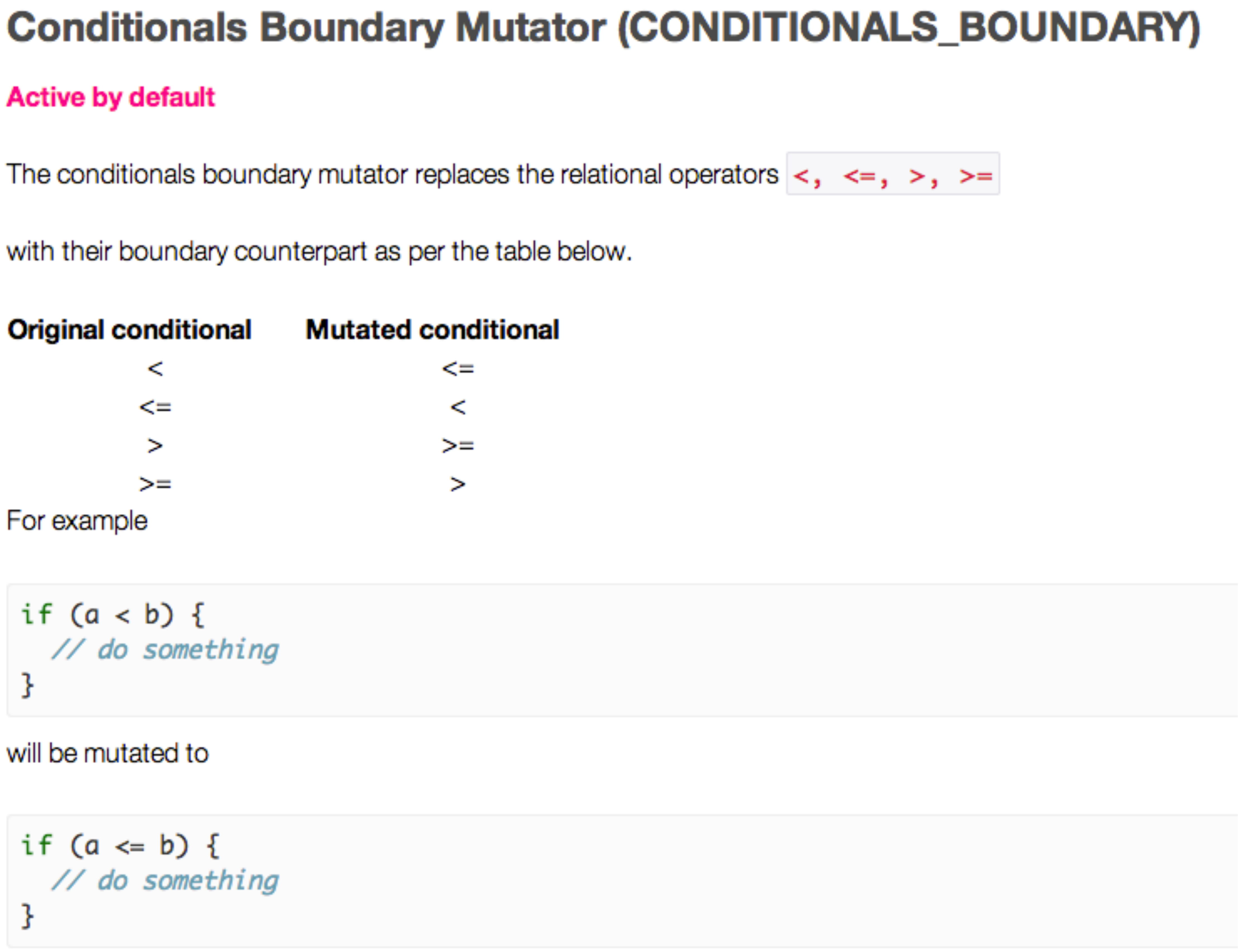
For example

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

will be mutated to

```
int a = b - c;
```

PIT operators



Conditionals Boundary Mutator (CONDITIONALS_BOUNDARY)

Active by default

The conditionals boundary mutator replaces the relational operators `<, <=, >, >=`

with their boundary counterpart as per the table below.

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

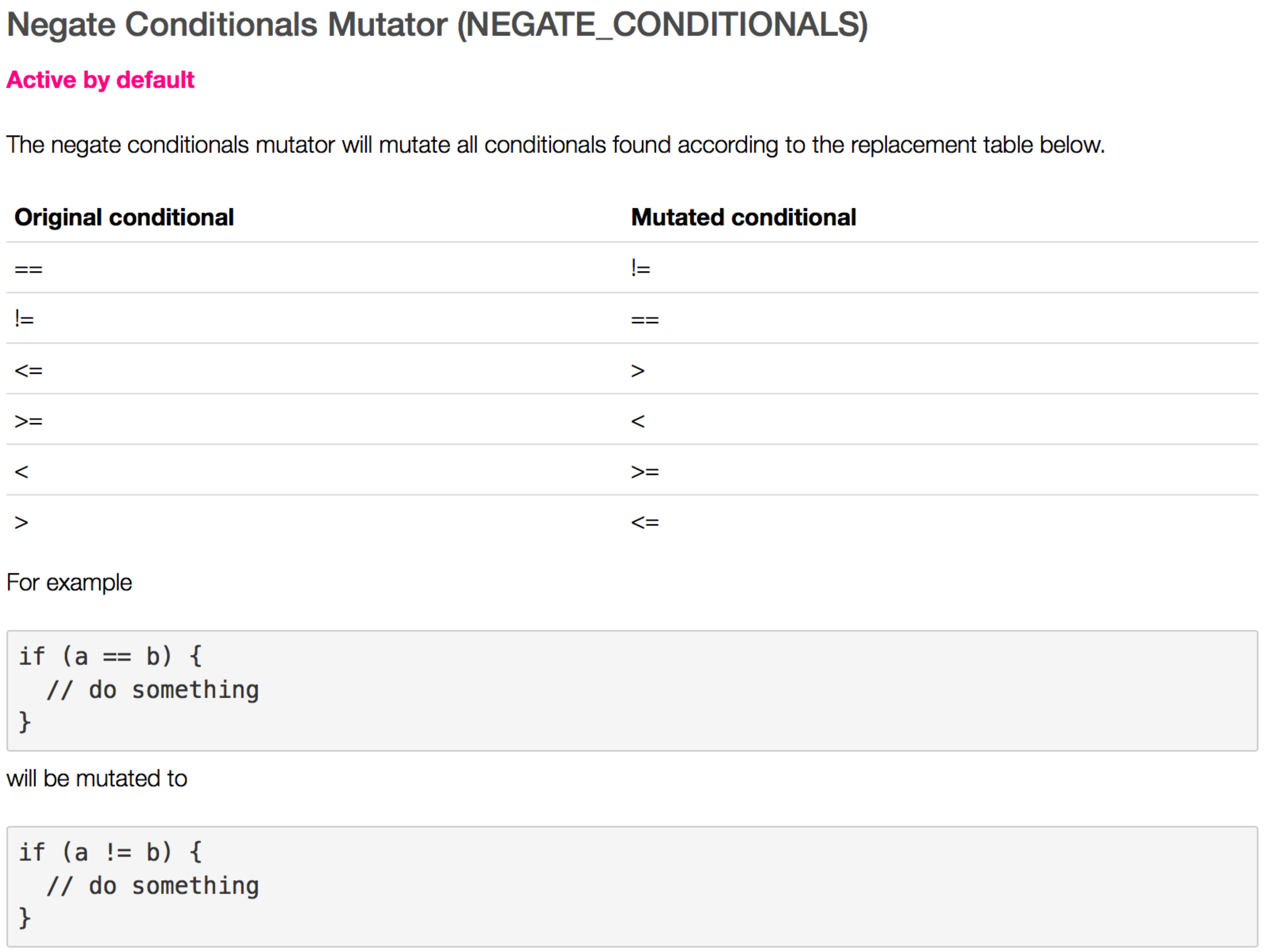
For example

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

will be mutated to

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

PIT operators



Negate Conditionals Mutator (NEGATE_CONDITIONALS)

Active by default

The negate conditionals mutator will mutate all conditionals found according to the replacement table below.

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

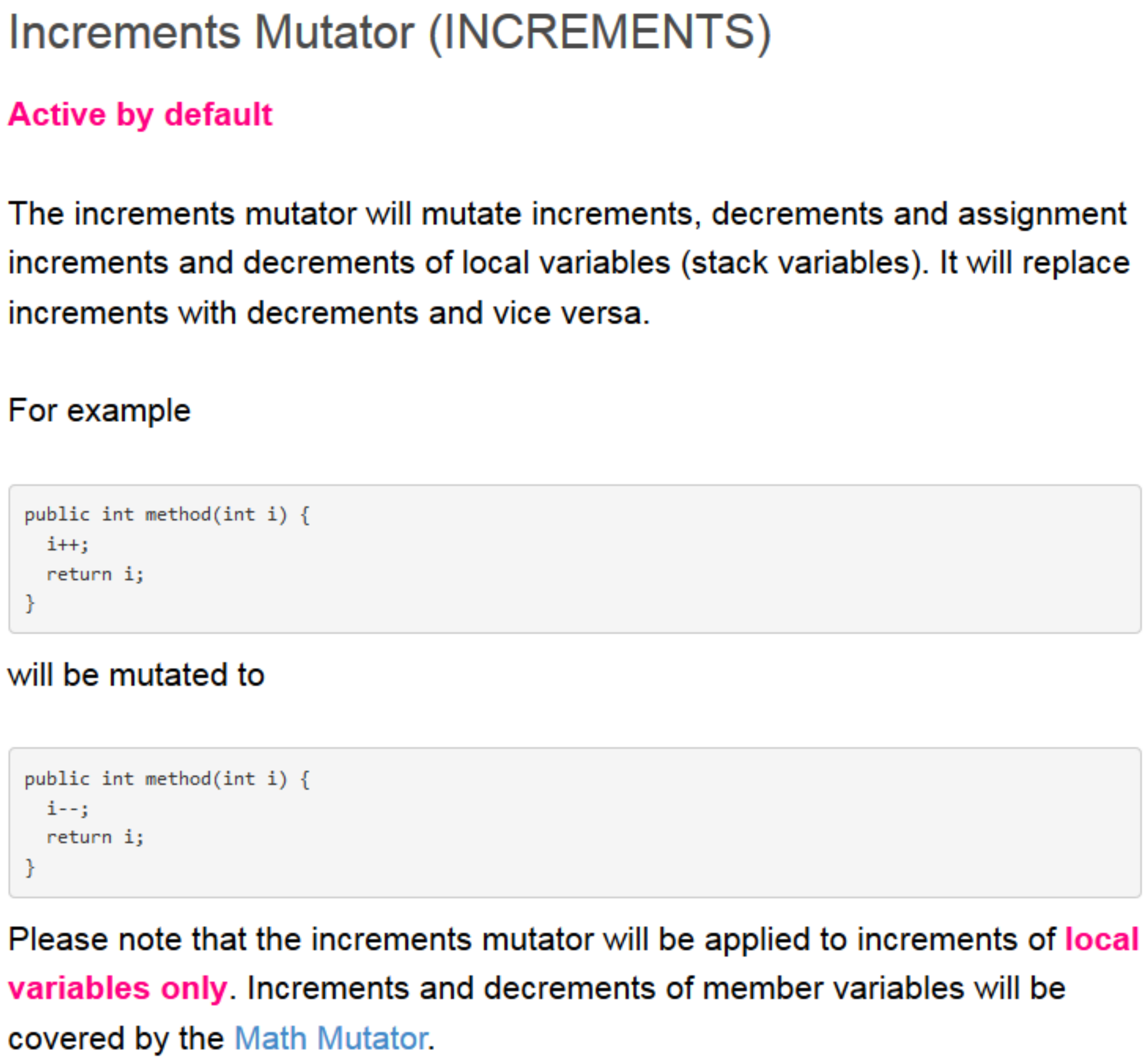
For example

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

will be mutated to

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

PIT operators



Increments Mutator (INCREMENTS)

Active by default

The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.

For example

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

will be mutated to

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

Please note that the increments mutator will be applied to increments of **local variables only**. Increments and decrements of member variables will be covered by the Math Mutator.

Exercise 2

```java
static int[] sum(int[] a, int[] b) {
  if (a == null || b == null || a.length != b.length)
    throw new IllegalArgumentException();
  int[] c = new int[a.length];
  for (int i = 0; i < a.length; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

1. Characterise all mutants obtained by applying the NEGATE_CONDITIONALS and CONDITIONALS_BOUNDARY operators. Are there syntactically invalid mutants?
2. Find tests that kill at least one valid mutant per operator at each code location. Are there any functionally-equivalent mutants?

Exercise 2

- CONDITIONALS_BOUNDARY mutant at the following location
  - `i < a.length` → `i <= a.length`
- NEGATE_CONDITIONALS mutants at
  - `a == null` → `a != null`
  - `b == null` → `b != null`
  - `a.length != b.length` → `a.length == b.length`
  - `i < a.length` → `i >= a.length`
- There are no syntactically invalid or functionally equivalent mutants
  - A syntactically invalid mutant would result if we mutated `a == null` → `a < null`
  - A functionally equivalent mutant would result if we mutated a
    `i < a.length` → `i != a.length`
  - PIT does not have these operations
- **Exercise 3**: Write tests that kill all PIT mutants for sum (Exercise 2).

Subsumption of other test criteria

- Program-based mutation can subsume other test criteria, through the use of appropriate mutation operators.
- Examples:
  - Node coverage (graph-based criteria): insert mutants at the start of each block.
  - Clause coverage (logic-based criteria): mutate conditional expressions [usually this will also mean edge coverage is satisfied].

Exercise 4

```java
public static int average(int[] v) {
  if (v == null)
    throw new IllegalArgumentException();
  int r = 0;
  if (v.length > 0) {
    for (int i = 0; i < v.length; i++)
      r = r + v[i];
    r = r / v.length;
  }
  return r;
}
```

1. Identify the basic blocks of method `average()` and draw its CFG
2. Identify mutants and tests to kill the mutants such that a) node coverage is satisfied by killing the mutants but **not** edge coverage, and b) no mutant results from mutating "`v.length > 0`".
3. To also achieve edge coverage, identify a mutation of "`v.length > 0`" and a test that kills the corresponding mutant.

The "bomb" strategy (A&O) for node coverage

```
public static int average(int[] v) {
  // m1: bomb();
  if (v == null) {
    // m2: bomb();
    throw new NullPointerException();
  }
  // m3: bomb();
  int r = 0;
  if (v.length > 0) {
    // m4: bomb();
    for (int i=0; i < v.length; i++) {
      // m5: bomb();
      r = r + v[i];
    }
    // m6: bomb();
    r = r / v.length;
  }
  // m7: bomb();
  return r;
}
```

```
static void bomb() {
  throw new Error();
}
```

- Operator that replace statements with "bombs" that immediately cause the program to terminate
- Node coverage requires each block in the program to be executed.
- To kill these mutants, we need to find test cases that reach each block.
- I.e., this mutation subsumes Node Coverage.

Exercise 5

```java
public static int daysInMonth(int m, int y) {
  if (m <= 0 || m > 12)
    throw new IllegalArgumentException("Invalid month: " + m);
  if (m == 2) {
    if (y % 400 == 0 || (y % 4 == 0 && y % 100 != 0))
      return 29;
    else
      return 28;
  }
  if (m <= 7) {
    if (m % 2 == 1)
      return 31;
    return 30;
  }
  if (m % 2 == 0)
    return 31;
  return 30;
}
```

1. Identify mutants defined by swapping || with && (and vice-versa).
2. Find tests that kill each mutant, if possible.

Mutation operators for integration testing

- **Integration mutants**: mutants are defined on the connections between components.
- Integration mutants may target component relations expressed by **method calls**, applying mutations to:
  - call sites
  - return statements
- … and may exploit **object-oriented features** such as:
  - inheritance
  - polymorphism

Integration Mutation

- Call deletion:
  - Integration Method Call Deletion (IMCD): "Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value."
  - **PIT**: Void Method Call Mutator / Non-void method call mutator
- Return value mutation:
  - Integration Return Expression Modification (IREM): "Each expression in each return statement in a method is modified".
  - **PIT**: Return Values Mutator
- Call argument mutation:
  - Integration Parameter Variable Replacement (IPVR): "Each parameter in a method call is replaced by another variable in the scope of the method call."
  - Integration Unary Operator Insertion (IUOI): "Each expression in a method call is modified by inserting all possible unary operators in front and behind it."
  - Integration Parameter Exchange (IPEX) "Each parameter in a method call is exchanged with each parameter of compatible types in that method call."

Void Method Call Mutator (VOID_METHOD_CALLS)

The void method call mutator removes method calls to void methods. For example

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

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

will be mutated to

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

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

Please note that constructor calls are **not considered void method calls**. See the Constructor Call Mutator for mutations of constructors or the Non Void Method Call Mutator for mutations of non void methods.

Return Values Mutator (RETURN_VALS)

The return values mutator mutates the return values of method calls. Depending on the return type of the method another mutation is used.[4]

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`
`float` `double`	replace the unmutated return value `x` with the result of `-(x+1.0)` if `x` is not `NAN` and replace `NAN` with `0`
`Object`	replace non-`null` return values with `null` and throw a `java.lang.RuntimeException` if the unmutated method would return `null`

For example

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

will be mutated to

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

Non Void Method Call Mutator (NON_VOID_METHOD_CALLS)

The non void method call mutator removes method calls to non void methods. Their return value is replaced by the Java Default Value for that specific type. See the table below.

Table: Java Default Values for Primitives and Reference Types

Type	Default Value
`boolean`	`false`
`int` `byte` `short` `long`	`0`
`float` `double`	`0.0`
`char`	`'\u0000'`
`Object`	

```
public int someNonVoidMethod() {
  return 5;
}

public void foo() {
  int i = someNonVoidMethod();
  // do more stuff with i
}
```

will be mutated to

```
public int someNonVoidMethod() {
  return 5;
}

public void foo() {
  int i = 0;
  // do more stuff with i
}
```

and for method calls returning an object type the call

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

public void foo() {
  Object o = someNonVoidMethod();
  // do more stuff with o
}
```

will be mutated to

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

public void foo() {
  Object o = null;
  // do more stuff with o
}
```

Exercise 6

```
public class Time implements Comparable<Time> {
  ...
  public int getHours()   { return hours;   }
  public int getMinutes() { return minutes; }
  ...
  @Override
  public int compareTo(Time t) {
    int c = getHours() - t.getHours();
    if (c == 0)
      c = getMinutes() - t.getMinutes();
    return c;
  }
  ...
}
```

Consider the PIT return-value and non-void method call mutators applied to getHours(), getMinutes() and compareTo()

1. Identify integration mutants based on the previous operators
2. For each mutant identify, if possible, a test case that exercises `compareTo` and:
   - **a)** kills the mutant;
   - **b)** weakly kills the mutant (i.e., produces an error but no failure);
   - **c)** reaches the mutation but does not cause an error;

Exercise 6.1 (solution)

operator	location	mutation
return-value mutator	`return hours`	`return hours == 0 ? 1 : 0`
	`return minutes`	`return minutes == 0 ? 1 : 0;`
	`return c;`	`return c == 0 ? 1 : 0;`
non-void call mutator	`c = getHours() - t.getHours()`	`c = 0 - t.getHours()`
		`c = getHours() - 0`
	`c = getMinutes() - t.getMinutes()`	`c = 0 - t.getMinutes()`
		`c = getMinutes() - 0`

Exercise 6.2 (possible solution)

mutation	**Not reach**	**c) reach but no infection**	**b) infection but no failure (weak kill)**	**a) kills**

`return hours == 0 ? 1 : 0`	Always reach	this = 00:00
		t = 00:00
			this = 02:02
			t = 01:01
				this = 02:00
				t = 01:00

`return minutes == 0 ? 1 : 0`	this = 02:00
	t = 01:00
			Infection always leads to failure	this = 00:02
				t = 00:01

`return c == 0 ? 1 : 0`	Always reach, infection and failure			this = 00:00
				t = 00:00

`c = getHours() - 0`	Always reach	Reach always leads to infection, since program flow is different (a method call is removed).	this = 00:00
			t = 00:00
				this = 02:00
				t = 01:00

`c = 0 - t.getHours()`				

`c = 0 - t.getMinutes()`	this = 02:00
	t = 01:00
				this = 00:02
				t = 00:01

`c = getMinutes() - 0`				

Exercise 7

```
public class Time {
  private int hours;
  private int minutes;
  public Time() { hours = 0; minutes = 0; }
  public Time(int hours, int minutes) {
    this.hours = hours;
    this.minutes = minutes;
  }
  public int getHours() { return hours; }
  public int getMinutes() { return minutes; }
  @Override
  public boolean equals(Object o) {
    if (o == this) {
      return true;
    }
    if (o instanceof Time) {
      Time t = (Time) o;
      return t.getHours() == getHours()
          && t.getMinutes() == getMinutes();
    }
    return false;
  }
}
```

- Consider the mutations obtained by:
  1. making the body of the first constructor empty
  2. removing the **this** keyword in the second constructor
  3. removing method equals()
- Whenever possible find test cases that kill each mutant.
- Are there functionally equivalent mutants?

Exercise 8

```
public class TimeWithSecs extends Time {
  private int seconds;
  public TimeWithSecs() { seconds = 0; }
  public TimeWithSecs(int hours, int minutes, int
seconds) {
    super(hours, minutes);
    this.seconds = seconds;
  }
  public int getSeconds() { return seconds; }
  @Override
  public boolean equals(Object o) {
    if (o == this) {
      return true;
    }
    if (o instanceof TimeWithSecs) {
      TimeWithSecs t = (TimeWithSecs) o;
      return t.getHours() == getHours()
          && t.getMinutes() == getMinutes()
          && t.getSeconds() == getSeconds();
    }
    return false;
  }
}
```

- Consider the mutations obtained by:
  1. deleting the initialisations of the seconds field of each constructor
  2. deleting the parent constructor invocation
  3. removing method equals()
- Whenever possible find test cases that kill each mutant.
- Are there functionally equivalent mutants?