Overview

PIT currently provides ten built-in mutators, of which seven are activated by default. The default set can be overridden, and different operators selected, by passing the names of the required operators to the `mutators` parameter.

Mutations are performed on the byte code generated by the compiler rather than on the source files. This approach has the advantage of being generally much faster and easier to incorporate into a build, but it can sometimes be difficult to simply describe how the mutation operators map to equivalent changes to a Java source file.

The operators are largely designed to be stable (i.e. not be too easy to detect) and minimise the number of equivalent mutations that they generate. Those operators that do not meet these requirements are not enabled by default.

Available mutators

We’ll briefly describe the available mutators in the following sections. See the test cases (http://code.google.com/p/pitestrunner/source/browse/pitest/src/test/java/org/pitest/mutationtest/engine/gregor/mutators/) for more detailed examples.

Here is the list of available mutators:

- Conditionals Boundary Mutator
- Negate Conditionals Mutator
- Remove Conditionals Mutator
- Math Mutator
- Increments Mutator
- Invert Negatives Mutator
- Inline Constant Mutator
- Return Values Mutator
- Void Method Calls Mutator
- Non Void Method Calls Mutator
- Constructor Calls Mutator
- Experimental Inline Constant Mutator
- Experimental Member Variable Mutator
- Experimental Switch Mutator

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.

<table>
<thead>
<tr>
<th>Original conditional</th>
<th>Mutated conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

For example

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

will be mutated to

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

**Negate Conditionals Mutator (NEGATE_CONDITIONALS)**

*Active by default*

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

<table>
<thead>
<tr>
<th>Original conditional</th>
<th>Mutated conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>!=</td>
</tr>
<tr>
<td>!=</td>
<td>==</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&lt;=</td>
</tr>
</tbody>
</table>

For example

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

will be mutated to

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

This mutator overlaps to a degree with the conditionals boundary mutator, but is less **stable** i.e these mutations are generally easier for a test suite to detect.

**Remove Conditionals Mutator (REMOVE_CONDITIONALS)**

The remove conditionals mutator will remove all conditionals statements such that the guarded statements always execute
For example

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

will be mutated to

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

Although not currently enabled by default it is highly recommended that you enable it if you wish to ensure your test suite has full coverage of conditional statements.

**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.

<table>
<thead>
<tr>
<th>Original operation</th>
<th>Mutated operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>*</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>*</code></td>
</tr>
<tr>
<td><code>%</code></td>
<td><code>*</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>`</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td><code>&gt;&gt;</code></td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td><code>&lt;&lt;</code></td>
</tr>
<tr>
<td><code>&gt;&gt;&gt;</code></td>
<td><code>&lt;&lt;</code></td>
</tr>
</tbody>
</table>

For example

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

will be mutated to

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

Keep in mind that the `+` operator on `String`s as in

```java
String a = "foo" + "bar";
```

is *not a mathematical operator* but a string concatenation and will be replaced by the compiler with something like

```java
String a = new StringBuilder("foo").append("bar").toString();
```
Please note that the compiler will also use binary arithmentic operations for increments, decrements and assignment increments and decrements of non-local variables (member variables) although a special `iinc` opcode for increments exists. This special opcode is restricted to local variables (also called stack variables) and cannot be used for member variables. That means the math mutator will also mutate

```java
public class A {
    private int i;

    public void foo() {
        this.i++;
    }
}
```

to

```java
public class A {
    private int i;

    public void foo() {
        this.i = this.i - 1;
    }
}
```

See the Increments Mutator for details.

**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

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

will be mutated to

```java
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.

**Invert Negatives Mutator (INVERT_NEGS)**

*Active by default*
The invert negatives mutator inverts negation of integer and floating point numbers. For example

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

will be mutated to

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

**Inline Constant Mutator (INLINE_CONSTS)**

The inline constant mutator mutates inline constants. An inline constant is a literal value assigned to a non-final variable, for example

```java
public void foo() {
    int i = 3;
    // do something with i
}
```

Depending on the type of the inline constant another mutation is used. The rules are a little complex due to the different ways that apparently similar Java statements are converted to byte code.

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>replace the unmutated value <code>true</code> with <code>false</code> and replace the unmutated value <code>false</code> with <code>true</code></td>
</tr>
<tr>
<td>integer, byte short</td>
<td>replace the unmutated value <code>1</code> with <code>0</code>, <code>-1</code> with <code>1</code>, <code>5</code> with <code>-1</code> or otherwise increment the unmutated value by one.</td>
</tr>
<tr>
<td>long</td>
<td>replace the unmutated value <code>1</code> with <code>0</code>, otherwise increment the unmutated value by one.</td>
</tr>
<tr>
<td>float</td>
<td>replace the unmutated values <code>1.0</code> and <code>2.0</code> with <code>0.0</code> and replace any other value with <code>1.0</code></td>
</tr>
<tr>
<td>double</td>
<td>replace the unmutated value <code>1.0</code> with <code>0.0</code> and replace any other value with <code>1.0</code></td>
</tr>
</tbody>
</table>

For example

```java
public int foo() {
    int i = 42;
    return i;
}
```

will be mutated to
public int foo() {
    int i = 43;
    return i;
}

Please note that the compiler might optimize the use of final variables (regardless whether those are stack variables or member variables). For example the following code

```java
public class A {
    private static final int VAR = 13;

    public String foo() {
        final int i = 42;
        return "" + VAR + ":" + i;
    }
}
```

will be changed/optimized by the compiler to

```java
public class A {
    public String foo() {
        return "13:42";
    }
}
```

In such situations the mutation engine can not mutate any variable.

**Return Values Mutator (RETURN_VALS)**

*Active by default*

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

<table>
<thead>
<tr>
<th>Return type</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>replace the unmutated return value <code>true</code> with <code>false</code> and replace the unmutated return value <code>false</code> with <code>true</code></td>
</tr>
<tr>
<td>integer, byte, short</td>
<td>if the unmutated return value is 0 return 1, otherwise mutate to return value 0</td>
</tr>
<tr>
<td>long</td>
<td>replace the unmutated return value <code>x</code> with the result of <code>x+1</code></td>
</tr>
<tr>
<td>float, double</td>
<td>replace the unmutated return value <code>x</code> with the result of (-(x+1.0)) if <code>x</code> is not <code>NAN</code> and replace <code>NAN</code> with 0</td>
</tr>
<tr>
<td>object reference</td>
<td>replace non-null return values with <code>null</code> and throw a <code>java.lang.RuntimeException</code> if the unmutated method would return <code>null</code></td>
</tr>
</tbody>
</table>

For example
```java
public Object foo() {
    return new Object();
}
```

will be mutated to

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

**Void Method Call Mutator (VOID_METHOD_CALLS)**

**Active by default**

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

```java
public void someVoidMethod(int i) {
    // does something
}
public int foo() {
    int i = 5;
    doSomething(i);
    return i;
}
```

will be mutated to

```java
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] [CONSTRUCTOR_CALL] for mutations of constructors or the [Non Void Method Call Mutator] [NON_VOID_METHOD_CALL] for mutations of non void methods.

**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.

*Java Default Values for Primitives and Reference Types*

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>integer, short, byte, long</td>
<td>0</td>
</tr>
</tbody>
</table>
float, double 0.0
char \u0000

Object reference null

For example

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

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

will be mutated to

```java
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

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

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

will be mutated to

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

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

Please note that this mutation is fairly unstable for some types (especially Objects where `NullPointerException` are likely) and may also create equivalent mutations if it replaces a method that already returns one of the default values without also having a side effect.
This mutator does not affect void methods or constructor calls. See Void Method Call Mutator[VOID_METHOD_CALL] for mutations of void methods and Constructor Call Mutator[CONSTRUCTOR_CALL] for mutations of constructors.

**Constructor Call Mutator (CONSTRUCTOR_CALLS)**

The constructor call mutator replaces constructor calls with `null` values. For example

```java
public Object foo() {
    Object o = new Object();
    return o;
}
```

will be mutated to

```java
public Object foo() {
    Object o = null;
    return o;
}
```

Please note that this mutation is fairly unstable and likely to cause `NullPointerExceptions` even with weak test suites.

This mutator does not affect non constructor method calls. See Void Method Call Mutator[VOID_METHOD_CALL] for mutations of void methods and Non Void Method Call Mutator[NON_VOID_METHOD_CALL] for mutations of non void methods.

**Experimental Inline Constant Mutator (EXPERIMENTAL_INLINE_CONSTS)**

The experimental inline constant mutator is similar to the inline constant mutator but attempts to provide a more consistent behaviour.

As much as possible the mutator simply adds 1 to the value. This is not done in some cases for due to esoteric JVM details. In the case of floating point types simply adding 1 would result in equivalent mutants (for large enough values, \((a + 1) == a\)).

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>replace the unmutated value <code>true</code> with <code>false</code> and replace the unmutated value <code>false</code> with <code>true</code></td>
</tr>
<tr>
<td>integer, byte short</td>
<td>replace the unmutated value <code>1</code> with <code>0</code>, <code>-1</code> with <code>0</code>, otherwise increment the unmutated value by one.</td>
</tr>
<tr>
<td>long</td>
<td>replace the unmutated value <code>1</code> with <code>0</code>, otherwise increment the unmutated value by one.</td>
</tr>
<tr>
<td>float</td>
<td>replace the <code>0.0</code> with <code>1.0</code>, <code>1.0</code> with <code>2.0</code> and replace any other value with <code>1.0</code></td>
</tr>
<tr>
<td>double</td>
<td>replace the <code>0.0</code> with <code>1.0</code>, <code>1.0</code> with <code>2.0</code>, any replace other value with <code>1.0</code></td>
</tr>
</tbody>
</table>
The rolling behaviour is a little surprising for shorts and ints as the example below shows

```java
short s = Byte.MAX_VALUE;
int i = Short.MAX_VALUE;
int j = Byte.MAX_VALUE;
```

will be mutated to

```java
short s = Byte.MIN_VALUE;
int i = Short.MIN_VALUE;
int j = Byte.MIN_VALUE;
```

**Experimental Member Variable Mutator (EXPERIMENTAL_MEMBER_VARIABLE)**

The experimental member variable mutator mutates classes by removing assignments to member variables. The mutator can even remove assignments to final members. The members will be initialized with their Java Default Value for the specific type. See the table below.

### Java Default Values for Primitives and Reference Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>integer, short, byte, long</td>
<td>0</td>
</tr>
<tr>
<td>float, double</td>
<td>0.0</td>
</tr>
<tr>
<td>char</td>
<td>'\u0000'</td>
</tr>
<tr>
<td>Object reference</td>
<td>null</td>
</tr>
</tbody>
</table>

For example

```java
public class MutateMe {
    private final int x = 5;
    ...
}
```

will be mutated to

```java
public class MutateMe {
    private final int x = 0;
    ...
}
```
Please Note: This mutator is likely to create equivalent mutations if a member variable is explicitly initialized with the Java default value for the specific type of the member variable as in

```java
public class EquivalentMutant {
    private int x = 0;
}
```

**Experimental Switch Mutator (EXPERIMENTAL_SWITCH)**

The switch mutator finds the first label within a switch statement that differs from the default label. It mutates the switch statement by replacing the default label (wherever it is used) with this label. All the other labels are replaced by the default one.

*Thanks to Stefan Penndorf who contributed this documentation.*

1. Integer numbers and booleans are actually represented in the same way be the JVM, it is therefore never safe if change a 0 to anything but a 1 or a 1 to anything but a 0.↩

2. Floating point numbers are always changed to 1 rather than adding 1 to the original value as this would result in equivalent mutations. Adding 1 to a large floating point number doesn’t necessarily change it’s value due to the imprecise way in which floats are represented.↩

3. See note above which applies to boths floats and doubles.↩

4. The strategy used by this mutator was translated from code in the Jumble project.↩