pitest.org

Real world mutation testing

PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started

User Group Issues Source Maven Central

Follow Tweet

What is mutation testing?

How it works in 51 words

Mutation testing is conceptually quite simple.

Faults (or 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.

What?

Really it is quite simple

To put it another way - PIT runs your unit tests against automatically modified versions of your application code. When the application code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail in this situation, it may indicate an issue with the test suite.

Why?

What's wrong with line coverage?

Traditional test coverage (i.e line, statement, branch, etc.) measures only which code is executed by your tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested.

The most extreme examples of the problem are tests with no assertions. Fortunately these are uncommon in most code bases. Much more common is code that is only partially tested by its suite. A suite that only partially tests code can still execute all its branches *(examples)*.

As it is actually able to detect whether each statement is meaningfully tested, mutation testing is the gold standard against which all other types of coverage are measured.

Why PIT?

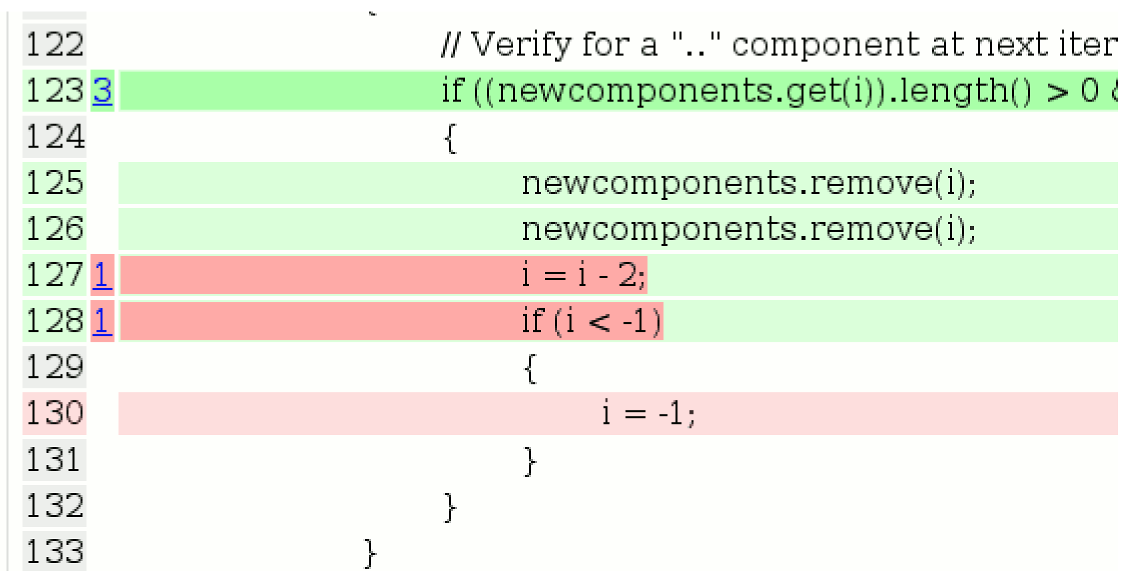
There are other mutation testing systems for Java, but they are not widely used.

They are mostly slow, difficult to use and written to meet the needs of academic research rather than real development teams.

PIT is different. It's

- fast - can analyse in minutes what would take earlier systems days
- easy to use - works with ant, maven, gradle and others
- actively developed
- actively supported

The reports produced by PIT are in an easy to read format combining line coverage and mutation coverage information.



```
                    // Verify for a ".." component at next iter
                    if ((newcomponents.get(i)).length() > 0 &
                    {
                        newcomponents.remove(i);
                        newcomponents.remove(i);
                        i = i - 2;
                        if (i < -1)
                        {
                            i = -1;
                        }
                    }
                }
```

Example snippet taken from coverage report of [Wicket Core]()

Light green shows line coverage, dark green shows mutation coverage.

Light pink show lack of line coverage, dark pink shows lack of mutation coverage.

Success stories

The Ladders

"... While we also used Clover for basic code coverage, as we got our PIT mutation coverage up into the 90s I stopped paying much attention to Clover"

"... This gave us extreme confidence in our tests ... The effects of that confidence were outstanding."

Kyle Winter, Lead Software Engineer, The Ladders

sky

". . . from my own personal experience of using PIT I've found it not only gives me confidence in the quality of both my own and others unit test quality, but has actually been a design aid in so much that as well as finding untested code, it can also find redundant code that when deleted still implements the intended functionality."

[more]()

Matt Kirk, Lead Developer, British Sky Broadcasting

Quickstart

Improve this page

Out of the box PIT can be launched from the command line, ant or maven. Third party components provide integration with Gradle, Eclipse, IntelliJ and others (see the links section for details).

The impatient can jump straight to the section for their chosen build tool - it may however be helpful to read the basic concepts section first.

Getting started

Maven quick start

Command line quick start

Ant quick start

Gradle quick start (external link)

More detail

Basic concepts

Available mutation operations

Incremental analysis

Advanced

Maven Quick Start

Improve this page

Installation

PIT is available from maven central since version 0.20.

Getting started

Add the plugin to build/plugins in your pom.xml

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
 </plugin>
```

That's it, you're up and running.

By default pitest will mutate all code in your project. You can limit which code is mutated and which tests are run using `targetClasses` and `targetTests`. Be sure to read the globs section if you want to use exact class names.

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>LATEST</version>
    <configuration>
        <targetClasses>
            <param>com.your.package.root.want.to.mutate*</param>
        </targetClasses>
        <targetTests>
            <param>com.your.package.root*</param>
        </targetTests>
    </configuration>
</plugin>
```

If no `targetClasses` are provided in versions before 1.2.0, pitest assumes that your classes live in a package matching your projects group id. In 1.2.0 and later verions pitest will scan your project to determine which classes are present.

PIT provides two goals

mutationCoverage goal

The mutation coverage goal analyses all classes in the codebase that match the target tests and target classes filters.

It can be run directly from the commandline

```
mvn org.pitest:pitest-maven:mutationCoverage
```

This will output an html report to target/pit-reports/YYYYMMDDHHMI.

To speed-up repeated analysis of the same codebase set the `withHistory` parameter to true.

```
mvn -DwithHistory org.pitest:pitest-maven:mutationCoverage
```

scmMutationCoverage goal

The scm mutation coverage goal analyses only classes that match the filters and the source file has a given status within the project source control system (by default ADDED or MODIFIED). This provides a quick way to check the coverage of changes prior to checking code in / pushing code to a repository.

```
mvn org.pitest:pitest-maven:scmMutationCoverage -Dinclude=ADDED,UNKNOWN -DmutationThreshold=85
```

To use this goal the maven scm plugin must be correctly configured for the project

This goal does not currently guarantee to analyse changes made to non public classes that are not inner classes.

Globs

Globs are pretty simple and will work as expected as long as you match packages (like `com.your.package.root.want.to.mutate*`). But if you match exact class names, inner classes won't be included. If you need them you'll have to either add a '*' at the end of the glob to also match them (`com.package.Class*` instead of `com.package.Class`) or to add another rule for it (`com.package.Class.*` in addition to `com.package.Class`).

Other options

PIT tries to work sensibly out of the box, but also provides many configuration options.

The number of threads and list of mutation operators are both worth having a play with.

reportsDirectory

Output directory for the reports

targetClasses

The classes to be mutated. This is expressed as a list of globs.

For example

```
<targetClasses>
    <param>com.mycompany.*</param>
</targetClasses>
```

or

```
<targetClasses>
    <param>com.mycompany.package.*</param>
    <param>com.mycompany.packageB.Foo*</param>
    <param>com.partner.*</param>
</targetClasses>
```

If no targetClasses are supplied pitest will automatically determine what to mutate.

Before 1.2.0 pitest assumed that all code lives in a package matching the maven group id. In 1.2.0 and later versions, the classes to mutate are determined by scanning the maven output directory.

targetTests

A list of globs can be supplied to this parameter to limit the tests available to be run.

This parameter can be used to point PIT to a top level suite or suites. Custom suites such as ClassPathSuite are supported. Tests found via these suites can also be limited by the distance filter (see below).

maxDependencyDistance

PIT can optionally apply an additional filter to the supplied tests, such that only tests a certain distance from a mutated class will be considered for running. e.g A test that directly calls a method on a mutated class has a distance of 1 , a test that calls a method on a class that uses the mutee as an implementation detail has a distance of 2 etc.

This filter will not work for tests that utilise classes via interfaces, reflection or other methods where the dependencies between classes cannot be determined from the byte code.

The distance filter is particularly useful when performing a targeted mutation test of a subset of classes within a large project as it avoids the overheads of calculating the times and coverage of tests that cannot exercise the mutees.

threads

The number of threads to use when mutation testing. By default a single thread will be used.

mutateStaticInitializers

Whether or not to create mutations in static initializers. Defaults to false.

mutators

List of mutation operators to apply.

for example

```
<configuration>
    <mutators>
        <mutator>CONSTRUCTOR_CALLS</mutator>
        <mutator>NON_VOID_METHOD_CALLS</mutator>
    </mutators>
</configuration>
```

For details of the available mutators and the default set applied see mutation operators.

excludedMethods

List of globs to match against method names. Methods matching the globs will be excluded from mutation.

excludedClasses

List of globs to match against class names. Matching classes will be excluded from mutation.

Prior to 1.3.0 matching test classes were also not run. From 1.3.0 onwards tests are excluded with the excludedTests parameter

excludedTestClasses

List of globs to match against test class names. Matching tests will not be run (note if a suite includes an excluded class, then it will "leak" back in).

avoidCallsTo

List of packages and classes which are to be considered outside the scope of mutation. Any lines of code containing calls to these classes will not be mutated.

If a list is not explicitly supplied then PIT will default to a list of common logging packages as follows

- java.util.logging
- org.apache.log4j
- org.slf4j
- org.apache.commons.logging

So, the configuration section must look like:

```
<avoidCallsTo>
    <avoidCallsTo>java.util.logging</avoidCallsTo>
    <avoidCallsTo>org.apache.log4j</avoidCallsTo>
    <avoidCallsTo>org.slf4j</avoidCallsTo>
    <avoidCallsTo>org.apache.commons.logging</avoidCallsTo>
</avoidCallsTo>
```

If the feature `FLOGCALL` is disabled, this parameter is ignored and logging calls are also mutated.

verbose

Output verbose logging. Defaults to off/false.

timeoutFactor

A factor to apply to the normal runtime of a test when considering if it is stuck in an infinite loop.

Defaults to 1.25

timeoutConstant

Constant amount of additional time to allow a test to run for (after the application of the timeoutFactor) before considering it to be stuck in an infinite loop.

Defaults to 4000

maxMutationsPerClass

The maximum number of mutations to create per class. Use 0 or -ve number to set no limit.

Defaults to 0 (unlimited)

jvmArgs

List of arguments to use when PIT launches child processes. This is most commonly used to increase the amount of memory available to the process, but may be used to pass any valid JVM argument.

For example when running on OpenJDK 7 the it is sometimes necessary to disable the split verifier.

```
<jvmArgs>
    <jvmArg>-XX:-UseSplitVerifier</jvmArg>
</jvmArgs>
```

jvm

The path to tha java executable to be used to launch test with. If none is supplied defaults to the one pointed to by `JAVA_HOME`.

outputFormats

List of formats in which to write mutation results as the mutations are analysed. Supported formats are HTML, XML, CSV.

Defaults to HTML.

failWhenNoMutations

Whether to throw error when no mutations found.

Defaults to true.

excludedGroups

List of TestNG groups or JUnit Categories to exclude from mutation analysis.

includedGroups

List of TestNG groups or JUnit Categories to include in mutation analysis.

mutationUnitSize

Maximum number of mutations to include in a single analysis unit.

Defaults to 0 (unlimited)

exportLineCoverage

Export line coverage data.

Defaults to false

mutationEngine

Engine to use when generating mutations. Additional engines may be added via plugins.

Defaults to gregor

testPlugin

The test framework to use. Support values are

- junit (default) - runs junit 3 and 4 tests
- testng - runs TestNG tests

Support for other test frameoworks such as junit5 can be added via plugins.

additionalClasspathElements

List of additional classpath entries to use when looking for tests and mutable code. These will be used in addition to the classpath with which PIT is launched.

detectInlinedCode

Enabled by default since 0.29.

Indicates if PIT should attempt to detect the inlined code generated by the java compiler in order to implement finally blocks. Each copy of the inlined code would normally be mutated separately, resulting in multiple identical looking mutations. When inlined code detection is enabled PIT will attempt to spot inlined code and create only a single mutation that mutates all affected instructions simultaneously.

The algorithm cannot easily distinguish between inlined copies of code, and genuine duplicate instructions on the same line within a finally block.

In the case of any doubt PIT will act cautiously and assume that the code is not inlined.

This will be detected as two separate inlined instructions

```
finally {
    int++;
    int++;
}
```

But this will look confusing so PIT will assume no in-lining is taking place.

```
finally {
    int++; int++;
}
```

This sort of pattern might not be common with integer addition, but things like string concatenation are likely to produce multiple similar instructions on the same line.

Defaults to false.

timestampedReports

By default PIT will create a date and time stamped folder for its output each it is run. This can can make automation difficult, so the behaviour can be suppressed by setting timestampedReports to false.

Defaults to true.

mutationThreshold

Mutation score threshold at which to fail build.

Please bear in mind that your build may contain equivalent mutations. Careful thought must therefore be given when selecting a threshold.

coverageThreshold

Line coverage threshold at which to fail build.

historyInputFile

Path to a file containing history information for incremental analysis.

historyOutputFile

Path to write history information for incremental analysis. May be the same as historyInputFile.

withHistory

Sets the history input and output files to point a project specific file within the temp directory.

This is a convenient way of using history files to speed up local analysis.

skip

You can skip the launch by adding the parameter `skip` on `configuration` section:

```
<configuration>
    <skip>true</skip>
</configuration>
```

It's very useful on maven module: when you need to skip an entire module, you can add this setting on the declaration of the plugin to ignore it.

Reporting Goal

Introduction

Starting with version 1.1.6, the pit maven plugin has a maven report goal. This goal should only be invoked as part of the maven site lifecycle. To execute this goal, the `mutationCoverage` goal must have already been executed to produce an HTML report (i.e. the `outputFormat` parameter must have HTML in it if the parameter is specified. The report goal then copies the latest HTML report to the site directory. If multiple reports exist (as in the case where `timestampedReports` is set to true), then only the report with the latest create time is used.

To generate the pit site report, set up the pitest-maven plugin in the project's pom as explained in the Getting Started section above and the `<reporting>` section as explained below. Then, execute both the `mutationCoverage` goal and the site lifecycle. For example:

```
mvn clean org.pitest:pitest-maven:mutationCoverage site
```

POM Configuration

The following configuration is the minimum required to generate the pit site report:

```xml
<reporting>
    <plugins>
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>LATEST</version>
            <reportSets>
                <reportSet>
                    <reports>
                        <report>report</report>
                    </reports>
                </reportSet>
            </reportSets>
        </plugin>
    </plugins>
</reporting>
```

Additional Parameters

Additional parameters exist to customize the generation of the report. They are:

skip

- Boolean indicating if the report generation should be skipped.
- Default is `false`
- User Property is `${pit.report.skip}`

reportsDirectory

- Indicates where the `mutationCoverage` goal wrote the pit HTML reports. This parameter does not need to be set unless the `reportsDirectory` parameter was set during the execution of the `mutationCoverage` goal. The value in this parameter must be an absolute path to the directory where the pit HTML report is located.
- Default is `${project.build.directory}/pit-reports`
- User property is `${reportsDirectory}`

sourceDataFormats

- List of strings specifying what data files should be read for the generation of the site report. Currently, the only supported value is "HTML" thus this parameter should not be used. Future versions of the pitest-maven plugin may implement other source data formats (i.e. XML or CSV).
- Default is "HTML"
- User property is `${pit.report.sourceDataFormats}`

siteReportName

- String that determines the name of the pit report that displays in the "Project Reports" section of the generated maven site.
- Default is "PIT Test Report"
- User property is `${pit.report.name}`

siteReportDescription

- String that determines the “Description” of the pit report in the “Project Reports” section of the generated maven site.
- Default is “Report of the pit test coverage”
- User property is `${pit.report.description}`

siteReportDirectory

- String that determines the name of the sub-directory under the directory where the maven site is written (usually target/site).
- Default value is “pit-reports” which means the pit report will be written to `target/site/pit-reports`
- User property is `${pit.report.outputdir}`

Example Showing All Options

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>LATEST</version>
            <configuration>
                <skip>false</skip>
                <reportsDirectory>${project.build.directory}/pit-custom-output-dir</reportsDirectory>
                <sourceDataFormats>
                    <sourceDataFormat>HTML</sourceDataFormat>
                </sourceDataFormats>
                <siteReportName>my-pit-report-name</siteReportName>
                <siteReportDescription>my pit report custom description</siteReportDescription>
                <siteReportDirectory>pit-custom-site-directory</siteReportDirectory>
            </configuration>
            <reportSets>
                <reportSet>
                    <reports>
                        <report>report</report>
                    </reports>
                </reportSet>
            </reportSets>
        </plugin>
    </plugins>
</reporting>
```

Handling projects composed of mutiple Maven modules (PitMP)

PitMP (PIT for Multi-module Project) is a Maven plugin to run PIT on multi-module projects.

By default, PIT mutates only the classes defined in the same module as the test suite.

Meanwhile, PitMP runs PIT on a complete project: the test suites are executed against the mutants generated in all classes of the project. In return, it produces a global mutation score for the project. The key rationale for the plugin is that some projects include test cases that are meant to assess the correctness of code regions that are in other modules.

PitMP extends PIT, it doesn’t rewrite PIT features. So, all PIT properties can be used. PitMP runs test suite as PIT does, just extending the list of classes to be mutated to the whole project tree, instead of mutating only the classes of the test suite module.

PitMP is availabe in Maven Central, and source and documentation are available in PitMP github.

Basic Concepts

Improve this page

Mutation Operators

PIT applies a configurable set of mutation operators (or mutators) to the byte code generated by compiling your code.

For example the `CONDITIONALS_BOUNDARY_MUTATOR` would modify the byte code generated by the statement

```
if ( i >= 0 ) {
    return "foo";
} else {
    return "bar";
}
```

To be equivalent to

```
if ( i > 0 ) {
    return "foo";
} else {
    return "bar";
}
```

PIT defines a number of these operations that will mutate the bytecode in various ways including removing methods calls, inverting logic statements, altering return values and more.

In order to do this PIT requires that the following debug information is present in the bytecode

- Line numbers
- Source file name

Most build systems enable this information by default.

Mutants

By applying the mutation operators PIT will generate a number (potentially a very large number) of mutants. These are Java classes which contain a mutation (or fault) which should make them behave differently from the unmutated class.

PIT will then run your tests using this mutant instead of the unmutated class. An effective set of tests should fail in the presence of the mutant.

Equivalent Mutations

Things are not quite this simple in practice as not all mutations will behave differently than the unmutated class. These mutants are referred to as equivalent mutations.

There are various reasons why a mutation might be equivalent including

- The resulting mutant behaves in exactly the same way as the original

For example, the following two statements are logically equivalent.

```
int i = 2;
if ( i >= 1 ) {
    return "foo";
}

//...
int i = 2;
if ( i > 1 ) {
    return "foo";
}
```

- The resulting mutant behaves differently but in a way that is outside the scope of testing.

A common example are mutations to code related to logging or debug. Most teams are not interested in testing these. PIT avoids generating this type of equivalent mutation by not generating mutations for lines that contain a call to common logging frameworks (this list of frameworks is configurable, to enable mutation of logging statements disable the feature `FLOGCALL`).

Running the tests

PIT runs your unit tests against the mutated code automatically. Before running the tests PIT performs a traditional line coverage analysis for the tests, then uses this data along with the timings of the tests to pick a set of test cases targeted at the mutated code.

This approach makes PIT much faster than previous mutation testing systems such as Jester and Jumble, and enables PIT to test entire code bases, rather than single classes at a time.

For each mutation PIT will report one of the following outcomes

- Killed
- Lived
- No coverage
- Non viable
- Timed Out
- Memory error
- Run error

Killed and Lived are self explanatory.

No coverage is the same as Lived except there were no tests that exercised the line of code where the mutation was created.

A mutation may time out if it causes an infinite loop, such as removing the increment from a counter in a for loop.

A non viable mutation is one that could not be loaded by the JVM as the bytecode was in some way invalid. PIT tries to minimise the number of non-viable mutations that it creates.

A memory error might occur as a result of a mutation that increases the amount of memory used by the system, or may be the result of the additional memory overhead required to repeatedly run your tests in the presence of mutations. If you see a large number of memory errors consider configuring more heap and permgen space for the tests.

A run error means something went wrong when trying to test the mutation. Certain types of non viable mutation can currently result in an run error. If you see a large number of run errors this is probably be an indication that something went wrong.

Under normal circumstances you should see no non viable mutations or run errors.

Overview

Improve this page

PIT currently provides some built-in mutators, of which most 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. To make configuration easier, some mutators are put together in groups. Passing the name of a group in the mutators parameter will activate all mutators of the group.

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 and groups

The following table list available mutators and whether or not they are part of a group :

Mutators	**"OLD_DEFAULTS" group**	**"DEFAULTS" group**	**"STRONGER" group**	**"ALL" group**
Conditionals Boundary	yes	yes	yes	yes
Increments	yes	yes	yes	yes
Invert Negatives	yes	yes	yes	yes
Math	yes	yes	yes	yes
Negate Conditionals	yes	yes	yes	yes
Return Values	yes			yes
Void Method Calls	yes	yes	yes	yes
Empty returns		yes	yes	yes
False Returns		yes	yes	yes
True returns		yes	yes	yes
Null returns		yes	yes	yes
Primitive returns		yes	yes	yes

Mutators	"OLD_DEFAULTS" group	"DEFAULTS" group	"STRONGER" group	"ALL" group
Remove Conditionals			EQ_ELSE case	yes
Experimental Switch			yes	yes
Inline Constant				yes
Constructor Calls				yes
Non Void Method Calls				yes
Remove Increments				yes
Experimental Argument Propagation				yes
Experimental Big Integer				yes
Experimental Member Variable				yes
Experimental Naked Receiver				yes
Negation				yes
Arithmetic Operator Replacement				yes
Arithmetic Operator Deletion				yes
Constant Replacement				yes
Bitwise Operator				yes
Relational Operator Replacement				yes
Unary Operator Insertion				yes

See the current code for current list (latest development version).

Default Mutators

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
}
```

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.

Invert Negatives Mutator (INVERT_NEGS)

Active by default

The invert negatives mutator inverts negation of integer and floating point numbers. For example

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

will be mutated to

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

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

For example

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

will be mutated to

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

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

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

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

```
String a = new StringBuilder("foo").append("bar").toString();
```

Please note that the compiler will also use binary arithmetic 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

```
public class A {
  private int i;

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

to

```
public class A {
  private int i;

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

See the Increments Mutator for details.

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
==	!=
!=	==
<=	>

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

For example

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

will be mutated to

```
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.

Return Values Mutator (RETURN_VALS)

This mutator has been superseded by the new returns mutator set. See Empty returns, False returns, True returns, Null returns and Primitive returns.

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;
}
```

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.

Void Method Call Mutator (VOID_METHOD_CALLS)

Active by default

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;
  someVoidMethod(i);
  return i;
}
```

will be mutated to

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

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

Empty returns Mutator (EMPTY_RETURNS)

Active by default

Replaces return values with an ‘empty’ value for that type as follows

- java.lang.String -> “”
- java.util.Optional -> Optional.empty()
- java.util.List -> Collections.emptyList()
- java.util.Collection -> Collections.emptyList()
- java.util.Set -> Collections.emptySet()

- java.lang.Integer -> 0
- java.lang.Short -> 0
- java.lang.Long -> 0
- java.lang.Character -> 0
- java.lang.Float -> 0
- java.lang.Double -> 0

Pitest will filter out equivalent mutations to methods that are already hard coded to return the empty value.

False returns Mutator (FALSE_RETURNS)

Active by default

Replaces primitive and boxed boolean return values with false.

Pitest will filter out equivalent mutations to methods that are already hard coded to return false.

True returns Mutator (TRUE_RETURNS)

Active by default

Replaces primitive and boxed boolean return values with true.

Pitest will filter out equivalent mutations to methods that are already hard coded to return true.

Null returns Mutator (NULL_RETURNS)

Active by default

Replaces return values with null. Methods that can be mutated by the EMPTY_RETURNS mutator or that are directly annotated with NotNull will not be mutated.

Pitest will filter out equivalent mutations to methods that are already hard coded to return null.

Primitive returns Mutator (PRIMITIVE_RETURNS)

Active by default

Replaces int, short, long, char, float and double return values with 0.

Pitest will filter out equivalent mutations to methods that are already hard coded to return 0.

Optional Mutators

Constructor Call Mutator (CONSTRUCTOR_CALLS)

Optional mutator that replaces constructor calls with `null` values. For example

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

will be mutated to

```
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 for mutations of void methods and Non Void Method Call Mutator for mutations of non void methods.

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

```
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.

Constant Type	Mutation
`boolean`	replace the unmutated value `true` with `false` and replace the unmutated value `false` with `true`
`integer` `byte` `short`	replace the unmutated value `1` with `0`, `-1` with `1`, `5` with `-1` or otherwise increment the unmutated value by one. [1]
`long`	replace the unmutated value `1` with `0`, otherwise increment the unmutated value by one.

Constant Type	Mutation
`float`	replace the unmutated values `1.0` and `2.0` with `0.0` and replace any other value with `1.0` [2]
`double`	replace the unmutated value `1.0` with `0.0` and replace any other value with `1.0` [3]

For example

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

will be mutated to

```java
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.

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`

Type	Default value
`int` `byte` `short` `long`	`0`
`float` `double`	`0.0`
`char`	`'\u0000'`
`Object`	`null`

For example

```
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
}
```

Please note that this mutation is fairly unstable for some types (especially Objects where `NullPointerExceptions` 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 for mutations of void methods and Constructor Call Mutator for mutations of constructors.

Remove Conditionals Mutator (REMOVE_CONDITIONALS)

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

For example

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

will be mutated to

```
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.

As shown above the basic remove conditionals mutator ensures that the statements following the conditional always execute. It will also only mutate only equality checks (eg ==, !=).

Additional specialised versions of the mutator exist that will ensure the block never executes so

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

will be mutated to

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

If an else block is present it will always execute

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

will be mutated to

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

Specialisations also exist that will mutate the bytecode instructions for order checks (eg <=, >).

The available specialisations are:

- REMOVE_CONDITIONALS_EQ_IF

- REMOVE_CONDITIONALS_EQ_ELSE
- REMOVE_CONDITIONALS_ORD_IF
- REMOVE_CONDITIONALS_ORD_ELSE

The names reflect which branch will be forced to execute (the “if” or the “else”) and the type of checks that will be mutated.

The reason these are not enabled by default is that there is a large degree of overlap in the tests required to kill these mutations and those required to kill mutations from other default operators such as the conditional boundaries mutator.

Remove Increments Mutator (REMOVE_INCREMENTS)

Optional mutator that removes local variable increments.

Experimental Mutators

Experimental Argument Propagation (EXPERIMENTAL_ARGUMENT_PROPAGATION)

Experimental mutator that replaces method call with one of its parameters of matching type.

Experimental Big Integer (EXPERIMENTAL_BIG_INTEGER)

Experimental mutator that swaps big integer methods.

Experimental Naked Receiver (EXPERIMENTAL_NAKED_RECEIVER)

Experimental mutator that replaces method call with a naked receiver.

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.

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`	`null`

For example

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

will be mutated to

```
  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

```
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.

Negation Mutator (ABS)

This mutator replace any use of a numeric variable (local valiable, field, array cell) with its negation. For example:

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

will be mutated to

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

Arithmetic Operator Replacement Mutator (AOR)

Like the math mutator, this mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The mutator is composed of 4 sub-mutators (AOR_1 to AOR_4) that mutate operators according to the table below.

Original operator	AOR_1	AOR_2	AOR_3	AOR_4
+	-	*	/	%
-	+	*	/	%
*	/	%	+	-
/	*	%	+	-
%	*	/	+	-

Arithmetic Operator Deletion Mutator (AOD)

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. For example

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

will be mutated to

```
int a = b;
```

and to

```
int a = c;
```

Constant Replacement Mutator (CRCR)

Like the inline constant mutator, this mutator mutates inline constant. The mutator is composed of 6 sub-mutators (CRCR1 to CRCR6) that mutate constants according to the table below.

Constant	CRCR1	CRCR2	CRCR3	CRCR4	CRCR5	CRCR6
c	1	0	-1	-c	c+1	c-1

Bitwise Operator Mutator (OBBN)

This mutator mutates bitwise and (&) and or (|). It is composed of three sub-mutators, OBBN1, OBBN2 and OBBN3 that respectively reverse the operators, replace a bitwise operation by its first member, and by its second member. For example

```
a & b;
```

will be mutated to

```
a | b;
```

by OBBN1, to

```
a;
```

by OBBN2 and to

```
b;
```

by OBBN3.

Relational Operator Replacement Mutator (ROR)

This mutator replaces a relational operator with another one. The mutator is composed of 5 sub-mutators (ROR1 to ROR5) that mutate the operators according to the table below.

Original operator	ROR1	ROR2	ROR3	ROR4	ROR5
<	<=	>	>=	==	!=
<=	<	>	>=	==	!=
>	<	<=	>=	==	!=
>=	<	<=	>	==	!=
==	<	<=	>	>=	!=
!=	<	<=	>	>=	==

Unary Operator Insertion (UOI)

This mutator inserts a unary operator (increment or decrement) to a variable call. It affects local variables, array variables, fields, and parameters. It is composed of 4 sub-mutators, UOI1 to UOI4 that insert operators according to the table below.

Variable	UOI1	UOI2	UOI3	UOI4

Variable	UOI1	UOI2	UOI3	UOI4
a	a++	a–	++a	–a

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 its value due to the imprecise way in which floats are represented. ↩
3. See note above which applies to both floats and doubles. ↩
4. The strategy used by this mutator was translated from code in the Jumble project ↩

FAQ

Improve this page

What does PIT stand for?

PIT began life as a spike to run JUnit tests in parallel, using separate classloaders to isolate static state. Once this was working it turned out to be a much less interesting problem than mutation testing which initially needed a lot of the same plumbing.

So PIT originally stood for Parallel Isolated Test. Now it stands for PIT.

What are the requirements for running PIT?

Since release 1.4.0 PIT requires Java 8 or above, earlier releases require Java 5. Either JUnit or TestNG must be on the classpath.

JUnit 4.6 or above is supported (note JUnit 3 tests can be run using JUnit 4 so JUnit 3 tests are supported). JUnit 5 is not supported out of the box, but a plugin can be found here

TestNG support in PIT is quite new. PIT is built and tested against TestNG 6.1.1, it may work with earlier and later versions but this has not yet been tested.

PIT is taking forever to run

Mutation testing is a computationally expensive process and can take quite some time depending on the size of your codebase and the quality and speed of your test suite. PIT is fast compared to other mutation testing systems, but that can still mean that things will take a while.

You may be able to speed things up by

- Using the CDG accelerator plugin
- Using more threads. The optimum number will vary, but will generally be between 1 and the number of CPUs on your machine.
- Limit the number of mutation per class. This will give you a less complete picture however.
- Use filters to target only those packages or classes that are currently of interest

One thing to watch out for that can slow PIT down are tests on the classpath that are not normally run. Some teams have very slow exhaustive tests or performance tests that are not run by their build scripts.

As PIT examines the entire classpath it will try to run these so may not even start running mutations for several hours. These tests can be excluded using the `excludedClasses` option.

The most effective way to use mutation testing is usually to limit analysis to code that you are changing. This strategy is discussed in a blog post.

Tooling is available to integrate pitest into pull requests.

PIT found no classes to mutate / no tests to run. What am I doing wrong?

This is most likely down to one of three issues

- Incorrect classpath
- Incorrect filters
- Incorrect mutable code path

Make sure that your code and tests are properly referenced on the classpath, and check that the filters are not excluding your code. If you have supplied a specific mutable code path, make sure it is correct.

Note that PIT is a bytecode mutator - it does not compile your code but instead modifies the byte code in memory. Your code must be on the classpath - PIT only requires the location of your source code in order to generate a human readable report.

My tests normally run green but PIT says the suite isn't green

Most commonly this is because either :

- PIT is picking up tests that are not included/are excluded in the normal test config
- Some test rely on an environment variable or other property set in the test config, but not set in the pitest config
- The tests have a hidden order dependency that is not revealed during the normal test run

If you are using an unusual or custom JUnit runner this can also sometimes causes problems. To make things fast PIT does some tricky stuff to split your tests into small independent units. This works well with most JUnit runners but if you encounter one where it doesn't please post to the user group.

Will PIT work with my mocking framework?

PIT is tested against the major mocking frameworks as part of its build.

PIT is currently the only mutation testing system known to work with all of JMock, EasyMock, Mockito, PowerMock and JMockit.

If your mocking framework of choice is not listed above the chances are still good that PIT will work with it. If it doesn't let us know and we'll look at getting that fixed.

My code has really poor test coverage, will mutation testing take forever?

No. Due to the way PIT picks which tests to run there is little or no execution time cost for mutations on lines that have no test coverage.

How does PIT choose which tests to run?

PIT chooses and prioritises tests based on three factors

- Line coverage
- Test execution speed
- Test naming convention

Per test case line coverage information is first gathered and all tests that do not exercise the mutated line of code are discarded. The remaining tests are then ordered by increasing execution time - test cases that belong to a class that is identified as a unit test for the mutated class are however weighted above other tests.

A class is considered to be the unit test for a particular class if it matches the standard JUnit naming convention of FooTest or TestFoo.

Unlike earlier systems PIT does not require that your tests follow this naming convention in order for it to work. Test names are used only as part of a heuristic to optimise run order.

I'm seeing a lot of timeouts, what's going on?

Timeouts when running mutation tests are caused by one of two things

- A mutation that causes an infinite loop
- PIT thinking an infinite loop has occurred but being wrong

In order to detect infinite loops PIT measures the normal execution time of each test without any mutations present. When the test is run in the presence of a mutation PIT checks that the test doesn't run for any longer than

normal time * x + y

Unfortunately the real world is more complex than this.

Test times can vary due to the order in which the tests are run. The first test in a class may have a execution time much higher than the others as the JVM will need to load the classes required for that test. This can be particularly pronounced in code that uses XML binding frameworks such as JAXB where classloading may take several seconds.

When PIT runs the tests against a mutation the order of the tests will be different. Tests that previously took milliseconds may now take seconds as they now carry the overhead of classloading. PIT may therefore incorrectly flag the mutation as causing an infinite loop.

An fix for this issue may be developed in a future version of PIT. In the meantime if you encounter a large number of timeouts, try increasing y in the equations above to a large value with –timeoutConst (timeoutConstant in maven).

I'm using OpenJDK 7 and keep getting java.lang.Verify errors

Java 7 introduced stricter requirements for verifying stack frames, which caused issues in earlier versions of PIT. It is believed that there were all resolved in 0.29.

If you see a verify error, please raise a defect. The issue can be worked around by passing -XX:-UseSplitVerifier to the child JVM processes that PIT launches using the jvmArgs option.

How does PIT compare the mutation testing system X

See mutation testing systems compared

I have mutations that are not killed but should be

Are the mutations in finally blocks? Do you seem to have two or more identical mutations, some killed and some not?

If so this is due to the way in which the java compiler handles finally blocks. Basically the compiler creates a copy of the contents of the finally block for each possible exit point. PIT creates separate mutations for each of the copied blocks. Most test suites are only able to kill one of these mutations.

As of 0.28 PIT contains experimental support for detecting inlined code that is now active by default.

Mutations in static initializers and enums

Static initializers and other code that is only run once per JVM (such as code in enum constructors) cause a bit of a problem with two of the strategies pitest uses to make mutation testing usable fast.

Coverage targeting

Pitest will only run tests that execute the line of code where a mutation is placed. Unfortunately the only test to execute a static initializer will be the first test to run that causes that class to load.

Mutant insertion

Pitest inserts mutants into a jvm by re-writing the class after it has loaded. This is orders of magnitude faster than starting a new jvm or creating a new classloader, but code in static initializer blocks is not re-run so the mutants have no effect.

Mitigation

Pitest tries to avoid mutating static initializer code. It will not create mutants in

- static initializers
- private methods called only from static initializers

You will however encounter other scenarios which this simple filtering will miss.

Can I activate more mutators without relisting all the default ones?

Yes. You can specify both individual mutators and groups of them using the same syntax.

Three groups are currently defined

- DEFAULTS
- STRONGER
- ALL

To use the defaults, plus some others

```
DEFAULTS, EXPERIMENTAL_MEMBER_VARIABLE
```

by the command line

or

```
<mutators>
  <mutator>DEFAULTS</mutator>
  <mutator>EXPERIMENTAL_MEMBER_VARIABLE</mutator>
</mutators>
```

in your pom.xml

Is it random?

No.

Given the same input pitest will always generate the same mutants, and (with a couple of caveats) will always produce the same results.

Pitest works hard to be fully deterministic, but two factors might cause the results to differ slightly between two runs with the same input.

Timeouts

Mutants causing infinite loops are detected by comparing the time taken to run a test without the mutant to the time taken when the mutant is present. Both these measurements can be affected by external factors (other processes on the machine etc etc), so a mutant may be detected as timed out on one run, but killed or surviving on another.

Static initializers

As discussed above static initalization code causes some problems for mutation testing, in certain circumstances it can also esult in small differences between runs, especially if timeouts occurs as these require starting a new jvm.

I have a problem, where can I get help?

Try asking a question at

http://groups.google.com/group/pitusers

Could I accidentally release mutated code if I use PIT?

No. The mutations that PIT generates are held in memory and never written to disk, except if explicitly enabled using the `EXPORT` feature. But even then they are only dumped inside the report directory and should not be released accidentally.

Where are snapshot releases uploaded to?

https://oss.sonatype.org/content/repositories/snapshots/org/pitest/

How do I use PIT with Gradle?

See PIT Gradle plugin

Is there any IDE integration?

Phil Glover maintains an Eclipse plugin. Pitclipse

Michal Jedynak maintains an IntelliJ plugin. PIT intellij plugin

It is also possible to launch PIT from most other IDEs as a Java application.

How can I combine all the reports for a project with multiple modules into a single report?

See Test Aggregation Across Modules

I'd like to help out, what can I do?

See how to help