# PIT: Test Coverage Tool Review
CA650 Assignment 2

Anthony Troy
14212116

April 22, 2016

**Requirements**

Given a chosen a software testing tool, either commercial or open source, produce a report that explains: the setup and usage of the tool; a sample run of the tool using its main functionality to set up and run three tests (code examples from your chosen language, screen shots etc.); an analysis of the coverage methods available using the tool (i.e. Graph, Logic, Input Space, Syntax); and an overall assessment of the usability, coverage and robustness of the tool. For instance, such a tool might be muJava. This project is due in simple A4 format (8 <= pages <= 14), accounts for 20% of the final module grade and must be submitted via Loop on April 22$^{\text{nd}}$ before 6 PM.

## Contents

## 1 Tool Overview

Our candidate test coverage tool, PIT, is an active mutation testing framework for Java and the JVM. Formely being named Parallel Isolated Test, the project's initial function was to isolate static state through running JUnit tests in parallel using separate classloaders, however the author later found that this turned out to be a "much less interesting problem than mutation testing which initially needed a lot of the same (code) plumbing" (**?**).

Mutation testing tools such as PIT are inherently linked to the domain of test coverage. A given test case's mutation adequacy score reflects the effectiveness of the case in detecting faults. This measurement of "testing tests" can be used to challenge the validity of one's traditional coverage metrics and in itself serve as a coverage criterion. As we will later explore, PIT supports reporting on both line coverage and mutation coverage information.

Mutation testing systems generally derive adequacy scores in four phases: mutant generation, test selection, mutant insertion and mutant detection. For most of the notable tools in this space these phases may be concurrent or hard to distinguish from each other. In less advanced systems only the generation phase is automated. Overall, from framework-to-framework the strategies adopted for these phases varies greatly. One reason for this disparity could be that many JVM-based mutation testing systems have been "written to meet the needs of academic research" (**?**).

The project positions itself as being a "state of the art mutation testing system" which provides "gold standard test coverage" for JVM-based languages. With respect to the existing tools in this space, PIT is argued to be quite performant (**??**). This is primarily due to fact that PIT adopts an effective byte code based approach in the mutant detection and generation phases. Generally systems will adopt either a source code or byte based approach at these stages.

One should note that using byte code during the mutation detection phase is quite computationally expensive, though this can be offset by an effective test selection phase. Additionally, systems using source code at the generation stage can also incur a large computational cost if a naive approach is followed. Tools like MuJava, Javalanche and Jumble too use byte code generation (**??**), however PIT performs various other optimisations. Rather than blindly running all cases against one mutation PIT will first determine overall line coverage and subsequently run only those cases that can actually reach the mutation. The system also supports an option to re-use the cached results from a previous analysis that can greatly reduce the cost of consecutive runs, which we will later demonstrate.

As previously highlighted, unlike the tools in this space stemming from academia, PIT is an open-source project that is commercially popular and actively contributed too. As you would expect, meeting this level of adoption requires being integratable the with the de facto testing and build tools in the JVM ecosystem. The framework has first-class support for the JUnit4, TestNg, Ant and Maven. One of course also has access to its command-line interface API, which in itself allows for extensibility. Third-party integrations with PIT have too been developed for tools such as Gradle, Eclipse and InteliJ (**?**). We later illustrate how to configure and run some of these integrations.

## 2 Mutation Testing

On a high-level, mutation testing can be described as a “fault-based testing technique” that provides a testing criterion which can be be used to measure the the effectiveness of a given test case in terms of its ability in detecting faults (**?**). Simply, faults are injected into one’s testing target, and when those tests are ran there are total and per case mutation adequacy scores derived. Under most systems, such as PIT, given those tests have failed then the mutation is killed and if they have passed then the mutation has survived. In turn, this total percentage of mutations killed acts as a testing criterion for test case quality.

The premise of mutation testing is intuitively appealing; discovering the input test data that can actually be used to reveal real faults. Nevertheless, it is not actually possible for any system to generate every possible permutation of mutant in a candidate program. Thus, as we will later illustrate with PIT, mutation testing systems will only offer a finite set of injectable mutators. Therefore, hopeful that these finite amount mutators will reveal faults, mutation testing is said to rely two assumptions: the Competent Programmer Hypothesis (CPH) and Coupling Effect (**?**).

The CPH asserts that a “program produced by a competent programmer is close to the final correct version of a program”. Thus, it is assumed that the source faults introduced by a competent programmer are relatively simple and can be corrected by small syntactical changes. Accordingly, in mutation testing only simple syntactical mutations are generally used. Contrarily, the Coupling Effect brings the suggestion that tests “capable of catching first order mutations will also detect higher order mutations that contain these first order mutations” (**?**).

As previously highlighted, the implementation of mutation testing systems are generally characterised in terms of their adopted mutation generation techniques. Another, and more high-level, distinction can be made with respect to how a given tool analyses how mutants are killed during the execution process. Such techniques can be classified into three types: strong, weak and firm mutation. Under strong mutation, or traditional mutation, “given program p, a mutant m of program p is said to be killed only if mutant m gives a different output from the original program p” (**?**).

In contrast, weak mutation requires only that the value produced at the point we introduce m is different from the corresponding value in p for m to be killed. This more optimal approach is adopted by PIT; instead of analysing each mutant after the execution of the entire suite run, mutants are checked immediately after their respective execution point. Firm mutation attempts to “overcome the disadvantages of both weak and strong mutations by providing a continuum of intermediate possibilities”. Although there is no tool yet available using this technique, **?** asserts that such would involve one outlining a ‘compare state’ threshold which would range “between the intermediate states after execution (weak) and the final output (strong)”.

# 3 Tool Evaluation

## 3.1 Included Documents

In conjunction with our evaluation, the following sources have been used and submitted with this report:

- README.md
- .gitignore
- practical/Dockerfile (builds and runs all of our review projects)
- practical/naive-app (SUT application)
- practical/maven-pit-review (source and build project under Maven)
- practical/ant-pit-review (source and build project under Ant)
- practical/gradle-pit-review (source and build project under Gradle)
- report/report.pdf
- report/report.tex
- report/report.bib

## 3.2 Tool Configuration

As you would expect, all of the build and run configurations in PIT are declarative rather than being programmatic. If one has the requirement to measure the effectiveness of a test suite through revealing mutation coverage, it is reasonable to expect that there should be no additional programming effort involved; PIT follows this philosophy. In fact the tool's JAR actually exposes no public APIs to be used within source code. There are only those APIs exposed for build and run configurations.

### 3.2.1 Build Configurations

As previously mentioned, PIT integrates with most of well-established JVM-based testing frameworks, this is true also for integrations with popular build tools. Firstly, we begin by building our Ubuntu 14 container host under test (CUT) to have Java 8 installed and configured:

```
FROM ubuntu:14.04
MAINTAINER Anthony Troy

### Linux deps
ENV DEBIAN_FRONTEND noninteractive
RUN set -ex \
  && sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.list \
  && apt-get update \
  && apt-get -y --force-yes install build-essential software-
    properties-common curl unzip
```

```
## Java dep
RUN set -ex \
  && echo oracle-java8-installer shared/accepted-oracle-license-v1
    -1 select true | debconf-set-selections \
  && add-apt-repository -y ppa:webupd8team/java \
  && apt-get update \
  && apt-get -y --force-yes install oracle-java8-installer \
  && rm -rf /var/lib/apt/lists/*
ENV JAVA_HOME /usr/lib/jvm/java-8-oracle
```

PIT's build tool support includes Maven, which is arguably the most popular automation framework for Java (?). We then install and configure the latest Maven on our CUT as follows:

```
### Maven dep
ENV MAVEN_VERSION 3.3.9
RUN mkdir -p /usr/share/maven \
  && curl -fsSL http://apache.osuosl.org/maven/maven-3/
    $MAVEN_VERSION/binaries/apache-maven-$MAVEN_VERSION-bin.tar.gz
    \
    | tar -xzC /usr/share/maven --strip-components=1 \
  && ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
ENV MAVEN_HOME /usr/share/maven
```

We then proceed to implement our subject under test (SUT) application. This naive application defines an interface allowing one to classify a triangle shape based on three supplied coordinates, wherein a classification can be either equilateral, isosceles, scalene or invalid. Accordingly we implement a TriangleType enum and following we define our Triangle class with one static method for classification. We also implement a TriangleTest case that meets total line coverage:

```
public enum TriangleType {
  INVALID, SCALENE, EQUILATERAL, ISOSCELES
}

public class Triangle {

  public static TriangleType classify(final int a, final int b,
    final int c) {
    int trian;
    if ((a <= 0) || (b <= 0) || (c <= 0)) {
      return TriangleType.INVALID;
    }
    trian = 0;
    if (a == b) {
      trian = trian + 1;
    }
    if (a == c) {
      trian = trian + 2;
    }
    if (b == c) {
      trian = trian + 3;
    }
    if (trian == 0) {
      if (((a + b) < c) || ((a + c) < b) || ((b + c) < a)) {
        return TriangleType.INVALID;
      } else {
        return TriangleType.SCALENE;
      }
    }
    if (trian > 3) {
      return TriangleType.EQUILATERAL;
    }
    if ((trian == 1) && ((a + b) > c)) {
      return TriangleType.ISOSCELES;
    } else if ((trian == 2) && ((a + c) > b)) {
      return TriangleType.ISOSCELES;
    } else if ((trian == 3) && ((b + c) > a)) {
      return TriangleType.ISOSCELES;
    }
    return TriangleType.INVALID;
  }

}


public class TriangleTest {

  @Test
  public void test1() {
    final TriangleType type = Triangle.classify(1, 2, 3);
    assertEquals(SCALENE, type);
  }

  @Test
  public void testInvalid1() {
    final TriangleType type = Triangle.classify(1, 2, 4);
    assertEquals(INVALID, type);
  }

  @Test
  public void testInvalid2() {
    final TriangleType type = Triangle.classify(1, 4, 2);
    assertEquals(INVALID, type);
  }

  @Test
  public void testInvalid3() {
    final TriangleType type = Triangle.classify(4, 1, 2);
    assertEquals(INVALID, type);

  }

  @Test
  public void testInvalidNeg1() {
    final TriangleType type = Triangle.classify(-1, 1, 1);
    assertEquals(INVALID, type);
  }

  @Test
  public void testInvalidNeg2() {
    final TriangleType type = Triangle.classify(1, -1, 1);
    assertEquals(INVALID, type);
  }

  @Test
  public void testInvalidNeg3() {
    final TriangleType type = Triangle.classify(1, 1, -1);
    assertEquals(INVALID, type);
  }

  @Test
  public void testEquiliteral() {
    final TriangleType type = Triangle.classify(1, 1, 1);
    assertEquals(EQUILATERAL, type);
  }

  @Test
  public void testIsoceles1() {
    final TriangleType type = Triangle.classify(2, 2, 3);
    assertEquals(ISOSCELES, type);
  }

  @Test
  public void testIsoceles2() {
    final TriangleType type = Triangle.classify(2, 3, 2);
    assertEquals(ISOSCELES, type);
  }

  @Test
  public void testIsoceles3() {
    final TriangleType type = Triangle.classify(3, 2, 2);
    assertEquals(ISOSCELES, type);
  }

  @Test
  public void testInvalid() {
    final TriangleType type = Triangle.classify(3, 1, 1);
    assertEquals(INVALID, type);
  }
}
```

With our SUT application in place we can begin our project configuration. All static configurations for Maven occur in a pom.xml file. Running PIT under Maven requires only having JUnit 4 or above in place along with registering the pitest−maven build plugin. After adding the minimal plugin configurations, PIT will now perform mutation testing at end the of the maven verfiy lifecycle phases (after test). We proceed by creating this file as follows:

```
   <repositories>
      <repository>
         <id>oss−sonatype</id>
         <name>oss−sonatype</name>
         <url>https://oss.sonatype.org/content/repositories/
  snapshots/</url>
         <snapshots>
            <enabled>true</enabled>
         </snapshots>
      </repository>
   </repositories>
   <pluginRepositories>
      <pluginRepository>
         <id>oss−sonatype</id>
         <name>oss−sonatype</name>
         <url>https://oss.sonatype.org/content/repositories/
  snapshots/</url>
         <snapshots>
            <enabled>true</enabled>
         </snapshots>
      </pluginRepository>
   </pluginRepositories>
   <dependencies>
      <dependency>
```

```
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.4</version>
            </plugin>
            <plugin>
                <groupId>org.pitest</groupId>
                <artifactId>pitest-maven</artifactId>
                <version>1.1.10-SNAPSHOT</version>
                <executions>
                    <execution>
                        <id>verify</id>
                        <phase>verify</phase>
                        <goals>
                            <goal>mutationCoverage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

PIT's also supports integration with Ant, a more lightweight offering to Maven (?). We proceed to install and configure the latest version of Ant on our CUT as follows:

```
### Ant dep
ENV ANT_VERSION 1.9.4
RUN set -ex \
  && cd \
  && wget -q http://archive.apache.org/dist/ant/binaries/apache-ant
    -${ANT_VERSION}-bin.tar.gz \
  && tar -xzf apache-ant-${ANT_VERSION}-bin.tar.gz \
  && mv apache-ant-${ANT_VERSION} /opt/ant \
  && rm apache-ant-${ANT_VERSION}-bin.tar.gz
ENV ANT_HOME /opt/ant
ENV PATH ${PATH}:/opt/ant/bin
```

Being simply a build 'toolkit', Ant does not have any support around package dependency management. Accordingly, we explicitly include our required JAR dependencies in our Ant project: `lib/junit-4.9.jar`, `lib/pitest-1.1.4.jar` and `lib/pitest-ant-1.1.4.jar`. Similar to Maven, all Ant project configuration mostly live within one file, `build.xml`. Running PIT under Ant requires somewhat more of a verbose configuration. We must: configure all source and test paths; define and configure a `pittest` task; configure the clean and tests tasks; and configure test reporting outputs and paths. With the following in place our `pit` task will run after the completion of the default `test` task:

```
    <path id="pitest.path">
        <pathelement location="lib/junit-4.9.jar" />
```

```
   <pathelement location="lib/pitest-1.1.4.jar" />
   <pathelement location="lib/pitest-ant-1.1.4.jar" />
</path>
<taskdef name="pitest" classname="org.pitest.ant.PitestTask"
 classpathref="pitest.path" />
<target name="clean">
   <delete dir="${classOutputDir}" />
</target>
<target name="compile" depends="clean">
   <mkdir dir="${classOutputDir}/classes" />
   <javac srcdir="src" includeantruntime="false" debug="true"
 debuglevel="source,lines" destdir="${classOutputDir}/classes" /
 >
</target>
<path id="test.path">
   <pathelement location="${classOutputDir}/classes" />
   <pathelement location="${classOutputDir}/test-classes" />
   <pathelement location="lib/junit-4.9.jar" />
</path>
<target name="test" depends="compile">
   <mkdir dir="${classOutputDir}/test-result" />
   <mkdir dir="${classOutputDir}/test-classes" />
   <javac includeantruntime="false" srcdir="test" destdir="${
 classOutputDir}/test-classes">
      <classpath refid="test.path" />
   </javac>
   <junit>
      <classpath refid="test.path" />
      <batchtest todir="${classOutputDir}/test-result">
         <fileset dir="test">
            <include name="**/*Test.java" />
         </fileset>
         <formatter type="xml" />
      </batchtest>
   </junit>
   <junitreport todir="${classOutputDir}/test-result">
      <fileset dir="${classOutputDir}/test-result">
         <include name="TEST-*.xml" />
      </fileset>
      <report format="frames" todir="${classOutputDir}/test-
 result/report" />
   </junitreport>
</target>
<target name="pit" depends="test">
   <path id="mutation.path">
      <path refid="pitest.path"/>
      <path refid="test.path"/>
   </path>
   <pitest pitClasspath="pitest.path" threads="2" classPath="
 mutation.path" targetTests="com.review.*" targetClasses="com.
 review.*" reportDir="pitReports" sourceDir="src" />
</target>
```

Third-party PIT support is also available for the emerging build framework Gradle, which builds upon the constructs available in both Maven and Ant. Under Gradle one configures a project through a Groovy-based DSL in a build.gradle file. As you would expect, this approach is much less verbose than that of Maven and Ant. A minimal PIT configuration simply requires: requiring JUnit as a dependency, specifying the third-party plugin; and defining a pitest target with one's targetClasses value. In turn, a Gradle pitest task will now be available for use. We implement such as follows:

```
plugins {
    id 'java'
    id "info.solidsoft.pitest" version "1.1.1"
}

sourceCompatibility = 1.8
version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
}

pitest {
    targetClasses = ['com.review.*']
    pitestVersion = "1.1.0"
}
```

### 3.2.2 Run Configurations

With our three configured projects in place we can now execute their respective mutation testing tasks. We add the SUT in the expected format to each project and begin testing as follows:

```
ENV WORKDIR /usr/src/app
RUN mkdir -p $WORKDIR/maven-pit-review
RUN mkdir -p $WORKDIR/ant-pit-review
RUN mkdir -p $WORKDIR/gradle-pit-review
WORKDIR $WORKDIR
VOLUME $WORKDIR

## Add Maven cache-order source
ADD maven-pit-review/pom.xml $WORKDIR/maven-pit-review
RUN cd maven-pit-review && mvn verify clean --fail-never
ADD naive-app/src $WORKDIR/maven-pit-review/src/main/java
ADD naive-app/test $WORKDIR/maven-pit-review/src/test/java
## Add Ant source
ADD ant-pit-review $WORKDIR/ant-pit-review/
ADD naive-app $WORKDIR/ant-pit-review/
## Add Gradle source
ADD gradle-pit-review $WORKDIR/gradle-pit-review/
ADD naive-app/src $WORKDIR/gradle-pit-review/src/main/java
ADD naive-app/test $WORKDIR/gradle-pit-review/src/test/java

## Compile then run Ant and Maven tests
RUN echo "Running naive-app PIT tests under Maven..."
RUN cd maven-pit-review && mvn verify
RUN echo "Running naive-app PIT tests under Ant..."
RUN cd ant-pit-review && ant pit
RUN echo "Running naive-app PIT tests under Gradle..."
RUN cd gradle-pit-review && gradle pitest
```

All PIT run configurations are respected regardless of the chosen build tool, thus, for the purpose of brevity we will continue our review under our Maven project. A variety of run configurations are supported, the most notable are:

```
<reportSets>
  <reportSet>
    <reports>
      <!-- outputs coverage-report to ./pit-reports -->
      <report>report</report>
    </reports>
  </reportSet>
</reportSets>
<configuration>
  <!-- whitelists our sources under test -->
  <targetClasses>
    <param>com.review.*</param>
  </targetClasses>
  <!-- whitelists our cases for test -->
  <targetTests>
    <param>com.review.*</param>
  </targetTests>
  <!-- amount of threads to use -->
  <threads>2</threads>
  <!-- whitelist of mutator operations to use -->
  <mutators>
    <!-- replace some relational operators -->
    <mutator>CONDITIONALS_BOUNDARY</mutator>
    <!-- negate all conditional operators -->
    <mutator>NEGATE_CONDITIONALS</mutator>
    <!-- remove all conditionals gaurding statements -->
    <mutator>REMOVE_CONDITIONALS</mutator>
    <!-- replaces binary arithmetic operations  -->
    <mutator>MATH</mutator>
    <!-- mutates increments & decrements -->
    <mutator>INCREMENTS</mutator>
    <!-- inverts negation of numbers -->
    <mutator>INVERT_NEGS</mutator>
    <!-- mutates inline constants  -->
    <mutator>INLINE_CONSTS</mutator>
    <!-- mutates the return values of method calls  -->
    <mutator>RETURN_VALS</mutator>
    <!-- removes method calls to void methods -->
    <mutator>VOID_METHOD_CALLS</mutator>
    <!-- removes method calls to non void methods -->
    <mutator>NON_VOID_METHOD_CALLS</mutator>
    <!-- replaces constructor calls with null values -->
    <mutator>CONSTRUCTOR_CALLS</mutator>
    <!-- removes class assignments to member variables -->
    <mutator>EXPERIMENTAL_MEMBER_VARIABLE</mutator>
    <!-- replaces the default switch statement label -->
    <mutator>EXPERIMENTAL_SWITCH</mutator>
  </mutators>
  <!-- whitelists packages out of scope for mutation -->
  <avoidCallsTo>
    <avoidCallsTo>java.util.logging</avoidCallsTo>
    <avoidCallsTo>org.apache.log4j</avoidCallsTo>
    <avoidCallsTo>org.slf4j</avoidCallsTo>
    <avoidCallsTo>org.apache.commons.logging</avoidCallsTo>
  </avoidCallsTo>
  <!-- enables verbose logging -->
  <verbose>true</verbose>
  <!-- uses CSV reporting -->
  <outputFormats>CSV</outputFormats>
  <!-- dont fail when no mutations are found -->
  <failWhenNoMutations>false</failWhenNoMutations>
  <!-- include line coverage in output -->
```

```
        <exportLineCoverage>true</exportLineCoverage>
        <!-- fail when coverage is less than 80% -->
        <coverageThreshold>80</coverageThreshold>
        <!-- where to store history -->
        <historyOutputFile>cache_dir/pit</historyOutputFile>
        <!-- where to load history from -->
        <historyInputFile>cache_dir/pit</historyInputFile>
      </configuration>
```

In the above list we have inverted or incremented most of the default configuration values. Outlined is the wide selection of common mutation operations PIT supports. When not following a whitelisting approach the following default mutators are applied for ease of use:

- CONDITIONALS_BOUNDARY_MUTATOR
- INCREMENTS_MUTATOR
- INVERT_NEGATIVES_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- RETURN_VALUES_MUTATOR
- VOID_METHOD_CALLS_MUTATOR

## 3.3 Test Coverage Run

Given the default run configurations against our SUT, upon building and running our host CUT we get the following output:

```
> $ cd practical && docker build -t pit . && docker run pit
          ....
[INFO] Adding org.pitest:pitest to SUT classpath
[INFO] Mutating from /usr/src/app/maven-pit-review/target/classes
[INFO] Defaulting to group id (com.review.app*)
PIT >> INFO : Sending 1 test classes to minion
PIT >> INFO : Sent tests to minion
PIT >> INFO : MINION : PIT >> INFO : Checking environment
PIT >> INFO : MINION : PIT >> INFO : Found  12 tests
PIT >> INFO : MINION : PIT >> INFO : Dependency analysis reduced
    number of potential tests by 0
PIT >> INFO : MINION : PIT >> INFO : 12 tests received
PIT >> INFO : Calculated coverage in 0 seconds.
PIT >> INFO : Created  1 mutation test units
PIT >> INFO : Completed in 3 seconds
             ....
================================================================================
- Statistics
================================================================================
>> Generated 44 mutations Killed 36 (82%)
>> Ran 115 tests (2.61 tests per mutation)
================================================================================
- Mutators
```

```
> org.pitest.mutationtest.engine.gregor.mutators.
    ConditionalsBoundaryMutator
>> Generated 10 Killed 2 (20%)
> KILLED 2 SURVIVED 8 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ReturnValsMutator
>> Generated 8 Killed 8 (100%)
> KILLED 8 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.MathMutator
>> Generated 9 Killed 9 (100%)
> KILLED 9 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.
    NegateConditionalsMutator
>> Generated 17 Killed 17 (100%)
> KILLED 17 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
```

We can note that for our naive application, which has total line coverage, 44 mutations are generated and only eight survive, thus resulting in a total mutation adequacy score of 82%. The seven default mutation operations are generated across our 12 test cases, resulting in the execution of 115 mutated test cases. The poor scoring of 20% on the CONDITIONALS_BOUNDARY_MUTATOR is unsurprising considering that our `classify` method is so branch-rich. We indeed have line coverage of these nine `if` statements but this scores proves we are performing the wrong test assertions.

## 4 Assessment

Mutation testing is argued to be the 'gold standard' of software testing. It is a well-established test criterion but an expensive one, as it involves generating and running many tests across several mutants. Indeed PIT is not the only mutation testing tool available for the JVM, however it is the most popular, active, performant and configurable tool in this space. PIT is not intended to be a replacement for code coverage systems, instead it can serve as a complementary toolkit. Simply put, in a straightforward and effective way PIT helps ensure that one is actually testing and verifying all the parts of the code that are assumed to be otherwise covered.