



Software Process Quality Assignment 2

Ruben Vasconcelos,
Student ID: 16212630,
Email: ruben.vasconcelos3@mail.dcu.ie,
Programme: MCM Part Time Year 2,
Module code: CA650.

April 12, 2018

PIT "state of the art mutation testing system".

Ruben Vasconcelos,

Student ID: 16212630,

Email: ruben.vasconcelos3@mail.dcu.ie

1 Introduction

In this report, I'm going to be evaluation PIT [1], a popular and widely used open source mutation testing tool for Java; in addition to a quick introduction of the mutation testing concept and overview of my practicum, in order to understand why I believe mutation testing can be beneficial.

2 Mutation testing (program based)

Mutation testing can be used as a means for measuring the adequacy or thoroughness of testing; in other words, mutation testing can be used to measure how good tests are, in addition to providing an indication of how well tests may perform as the software under evaluation evolves, throughout its lifecycle.

Typical unit testing aims to expose unexpected behaviour through failed tests that emulate software execution; mutation testing takes an interesting approach in the sense that it looks for the opposite; defects are introduced into the software under test (this is known as a mutant), the tests are run and if they fail then that is said to have killed the mutant (passed). Mutation testing instead outlines the cases where no failure occurs and mutation coverage or mutation score is simply the ratio of the killed mutants over the total generated number.

Mutation testing is a great way to detect faults, in fact, it is the strongest test criterion for finding faults according to Paul Ammann and Jeff Offutt [2]; mutation testing's main drawback is the fact that it can be quite expensive. Mutation testing closely relates to syntax coverage, mutants are generated based on mutant operators, which are simple syntactic rules that modify the software's syntax (more on PIT's operators further down). Picking an adequate set of operators is extremely important, trying to cover too many will affect scalability and picking too few will affect their effectiveness; therefore, mutation tools require a comprehensive set of mutants, that can easily be extended in order to keep it practical without compromising effectiveness.

3 Practicum

Recent developments in the area of container technologies such as Docker[3] have fuelled great interest from software development companies in PaaS clouds; in fact, most companies successfully using containers in production end up building their own sort of internal PaaS.

The benefits of using container technologies are well documented in many high-quality research papers and success stories from larger companies. The problem lies in the way a container based PaaS is set up, as it changes on a company basis often involving large amounts of scripting; the lack of standardization in addition to smaller budgets makes it harder for smaller companies trying to adopt the same tools. The main focus of my practicum is trying to solve this issue by offering a standard software development driven approach for creating such platforms, by breaking platforms into this notion of modules and components:

- Module: General overview and requirements for things such as infrastructure, logging and monitoring. The application knows how to run modules and modules offer a layer of abstraction on top of the vendor specific tools (components).
- Components: Are tool specific, for example, two infrastructure components could be Terraform[4] and Cloudformation[5], both can be used for building a platform's infrastructure. The general assumption is that the code/scripting required for running either is almost identical, as is the final result. So a test that validates one component could be reused to validate the other assuming both define identical platforms.

This would allow users to share their code for building platforms and the tests required for validating them via a central location (hub or server), that is being developed as part of the tool. This is where mutation testing could come in handy. Initially, I planned on using a user rating system to make it easier to find high-quality modules/components and even though that approach has some benefits, it is flawed in the way that popular tools will attract more users and hence receive better ratings. I had been searching for a solution to provide users with a more scientific measurement; so I intend to use mutation testing to give users an indication of how adequate the tests are for particular modules/components.

4 PIT

4.1 Intro

PIT is an easy to use open source mutation testing tool for Java, PIT does not offer as many features as some of the more popular Java mutation testing tools that were designed for research such as MuJava [6] and Major [7], but it makes up for it because of the fact that it is actively maintained, integrates well with development tools such as Ant or Maven and it can be invoked via the command line (more on this further down the paper), this makes it more appealing to software development companies. Some of PIT's main advantages are its scalability and support of mutant operators that conform with the current state of the art mutation research.

PIT operates on bytecode and optimises mutant execution by running only the test that could kill a mutant, making it very fast; in addition, it has a very low memory overhead because it does not require any program to be written on the disk, by keeping one mutant in memory at a time [8].

Personally, I would prefer if it took a similar approach to MuJava and saved mutants to disk, or the very least offered a feature that allowed for a side by side comparison in order to make it easier to examine the results, but I can see the advantages of not doing so in a continuous integration world; although, it is fairly straightforward to deduce the mutation in most cases due to its good documentation on operators.

4.2 Setup

Setting up PIT using development tools such as Maven is relatively easy, simply create a Java package with your code and a pom.xml file (see figure 1, a). Configuring Maven can be somewhat intimidating at first due to the number of possible configurations, but the documentation and resources available online are of good quality; in order to use PIT as part of your Maven project all you need to do is add a new plugin to the pom.xml file (see figure 1, b). The PIT plugin also comes with a very useful verbose option that can be used to help debug setup issues.



Figure 1: PIT setup using Maven.

4.3 PIT Mutant Generation

Mutant generation in PIT is a two-stage process, there is an initial scan over all classes in order to identify possible mutations points, also known as MutationIdentifiers (consist of the location and the name of the operator used to recreate mutants). The mutated bytecode is generated during the first stage, but interestingly enough it gets immediately discarded and only the MutationIdentifiers are stored in memory, the reason for this is: so large numbers of mutants information can be stored in memory. Eventually, each mutant gets executed in their own child JVM processes, at which point the bytecode is regenerated.

Child JVM process are used because often tests may leave the JVM in a different state to the initial. Creating child JVM processes is expensive but pit offers some configuration options for controlling the number of mutants that get executed per child JVM [8].

PIT supports a rather small number of operators in comparison to some of the other mutation testing tools, I guess the idea is to limit the number of mutants in order to reduce execution time [9]. It's a fact that mutation testing is a very expensive form of testing, PIT is obviously trying to maximize the benefits of mutation testing while trying to somewhat reduce its cost; the downside as pointed out by previous studies is that it could result in a set of low-quality mutants [10].

Active by default	Deactivated by default
Conditionals Boundary	Constructor Calls
Increments	Inline Constant
Invert Negatives	Non Void Method Calls
Math	Remove Conditionals
Negate Conditionals	Experimental Member variable
Return Value	Experimental Switch
Void Method Calls	

It's very easy to configure the mutation operators that get used, all that you have to do is simply update the pom.xml file with the list of desired operators (see figure 3).

```

</targetTests>
<!-- <verbose>true</verbose> -->
<mutators>
  <mutator>MUTATOR NAME</mutator>
</mutators>
</configuration>
</plugin>

```

Figure 2: PIT Maven plugin.

4.4 Usage

Onto the fun part now, technically my practicum could support components written in many languages and I haven't been able to find a tool that would give consistent results across different languages, so I decided to use part of an assignment I wrote for a different module in order to try out PIT. The code consists of 5 functions (see figure 3):

```

public class CryptoAnalysis {
    private int L0, R0, L4, R4;
    private int num_pairs = 200;
    private String plaintext[] = new String[num_pairs];
    private String cyphertext[] = new String[num_pairs];

    int getBit(int num, int n) { return (num >> (31-n)) & 1; }

    void splitPairs(int wordIndex) {
        L0 = (int) Long.parseLong(plaintext[wordIndex].substring(0,8), 16);
        R0 = (int) Long.parseLong(plaintext[wordIndex].substring(8, 16));
        L4 = (int) Long.parseLong(cyphertext[wordIndex].substring(0,8), 16);
        R4 = (int) Long.parseLong(cyphertext[wordIndex].substring(8, 16));
    }

    void readKnownTextPairs(String fileName) throws IOException {
        BufferedReader bufferedReader = new BufferedReader(new FileReader(fileName));
        int count = 0;
        boolean isPlainText = true;
        String line = bufferedReader.readLine();

        try {
            while(line != null && count < plaintext.length) {
                if(line.length() != 0) {
                    if(isPlainText) {
                        plaintext[count] = line.substring(12);
                    }
                    else {
                        cyphertext[count] = line.substring(12);
                        count++;
                    }
                    isPlainText = !isPlainText;
                }
                line = bufferedReader.readLine();
            }
            bufferedReader.close();
        }
        catch(IOException e) { System.out.println("Oh Oh failed to read input file: IOException"); throw e; }
    }

    int [] getSingleSplit() {
        int [] split = {L0, R0, L4, R4};
        return split;
    }

    String [][] getTextPairs() {
        String [][] pairs = {plaintext, cyphertext};
        return pairs;
    }
}

```

Figure 3: Source code under test.

- **getBit**: gets the nth bit of a number.
- **readKnownTextPairs**: read 200 plaintext/ciphertext pairs from a given file.
- **splitPairs**: split each pair of ciphertext and plaintext into two halves.
- **getSingleSplit**: returns the left and right half of a single know text pair
- **getTextPairs**: return the all text pairs.

Pit makes use of the JUnit framework as the basis for the tests required to kill the mutants. I decided to start with two very basic tests (see figure 4), then run the mutation test framework and improve the tests from there on based on the mutation coverage results.

```
@Test
public void test_read_pairs() {
    CryptoAnalysis crypto = new CryptoAnalysis();
    try {
        crypto.readKnownTextPairs("/app/src/main/java/known.txt");

        String [][] pairs = crypto.getTextPairs();

        assertEquals(pairs[0][199], "aea129c37cc07d12");
        assertEquals(pairs[1][199], "c4d88584deb8c60");
    }
    catch(IOException e) {
        fail("Did not expect an Exception");
    }
}

@Test (expected = IOException.class)
public void test_read_file_exception() throws IOException {
    CryptoAnalysis crypto = new CryptoAnalysis();
    crypto.readKnownTextPairs("invalid.txt");
}
```

Figure 4: Initial tests.

In order to run the unit tests you can use the command **mvn test** this will also compile any of the required files for the mutation tests; afterwards, you can then run PIT using the command **mvn pitest:mutationCoverage**, which should print a general summary log (see figure 5).

<pre>- Timings ===== > scan classpath : < 1 second > coverage and dependency analysis : < 1 second > build mutation tests : < 1 second > run mutation analysis : < 1 second > Total : 1 seconds - Statistics ===== >> Generated 15 mutations Killed 4 (27%) >> Ran 11 tests (0.73 tests per mutation) - Mutators ===== > org.pitest.mutationtest.engine.gregor.mutators.ConditionalBoundaryMutator >> Generated 1 Killed 0 (0%) > KILLED 0 SURVIVED 1 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.IncrementsMutator >> Generated 1 Killed 1 (100%) > KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0 > MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0 > NO_COVERAGE 0</pre>	<pre>> org.pitest.mutationtest.engine.gregor.mutators.VoidMethodCallMutator >> Generated 2 Killed 0 (0%) > KILLED 0 SURVIVED 2 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 3 Killed 1 (33%) > KILLED 1 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0 > MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0 > NO_COVERAGE 2 > org.pitest.mutationtest.engine.gregor.mutators.MathMutator >> Generated 3 Killed 0 (0%) > KILLED 0 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0 > MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0 > NO_COVERAGE 3 > org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator >> Generated 5 Killed 2 (40%) > KILLED 2 SURVIVED 3 TIMED_OUT 0 NON_VIABLE 0 > MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0 > NO_COVERAGE 0 [INFO] [INFO] BUILD SUCCESS [INFO]</pre>
--	---

(a) First Half.

(b) Second Half.

Figure 5: PIT run output.

4.5 Coverage

PIT generates an HTML report that can be used to deduce the mutants, the initial page when you open the report shows the overall line coverage and the mutation score (see figure 6). Selecting the class name will redirect to a page with a more detailed report.

Pit Test Coverage Report

Package Summary

default

Number of Classes	Line Coverage	Mutation Coverage
1	72% <div><div></div><div>21/29</div></div>	27% <div><div></div><div>4/15</div></div>

Breakdown by Class

Name	Line Coverage	Mutation Coverage
CryptoAnalysis.java	72% <div><div></div><div>21/29</div></div>	27% <div><div></div><div>4/15</div></div>

Figure 6: PIT report landing page.

Reports generated with earlier versions of PIT show each line code in their reports, but in more recent versions that view was discarded and instead the line number is presented in blue on the left-hand side of the mutant list for that particular line (see figure 7).

Mutations	
1.	Replaced integer subtraction with addition → NO_COVERAGE
2.	Replaced Shift Right with Shift Left → NO_COVERAGE
3.	Replaced bitwise AND with OR → NO_COVERAGE
4.	replaced return of integer sized value with (x == 0 ? 1 : 0) → NO_COVERAGE
1.	changed conditional boundary → SURVIVED
2.	negated conditional → SURVIVED
3.	negated conditional → SURVIVED
1.	negated conditional → KILLED
1.	negated conditional → KILLED
1.	Changed increment from 1 to -1 → KILLED
1.	negated conditional → SURVIVED
1.	removed call to java/io/BufferedReader::close → SURVIVED
1.	removed call to java/io/PrintStream::println → SURVIVED
1.	mutated return of Object value for CryptoAnalysis::getSingleSplit to (if (x != null) null else throw new RuntimeException) → NO_COVERAGE
1.	mutated return of Object value for CryptoAnalysis::getTextPairs to (if (x != null) null else throw new RuntimeException) → KILLED

Figure 7: PIT report - Mutations

For the mutations list we can see that the number of mutants generated by PIT was 4, 3 and 1 for lines 9, 25 and 26 respectively.

- The white background means that the mutants were never executed by the tests.

- The red or dark pink background means that the mutants were executed but not killed.
- The green background means that the mutants were executed and successfully killed.

Bellow the mutations view there is a list of active mutators and unit tests executed (see figure 8).

Active mutators

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

Tests examined

- TestCryptoAnalysis.test_expected_behaviour(TestCryptoAnalysis) (107 ms)
- TestCryptoAnalysis.test_read_file_exception(TestCryptoAnalysis) (1 ms)

Figure 8: PIT report - Active operators.

The next step is to try to create tests that execute the mutants with a white background from figure 7 (not covered), I went ahead and created two new tests (see figure 9).



```

@Test
public void test_split_pairs() {
    CryptoAnalysis crypto = new CryptoAnalysis();
    try {
        crypto.readKnownTextPairs("/app/src/main/java/known.txt");
        crypto.splitPairs(0);

        int [] split = crypto.getSingleSplit();

        assertEquals(split[0], -1477322454);
        assertEquals(split[1], -2100766466);
        assertEquals(split[2], -1078643750);
        assertEquals(split[3], -945344036);
    }
    catch(IOException e) {
        fail("Did not expect an Exception");
    }
}

```

```

@Test
public void test_get_n_bit() {
    CryptoAnalysis crypto = new CryptoAnalysis();

    assertEquals(crypto.getBit(-1477322454, 0), 1);
    assertEquals(crypto.getBit(0, 31), 0);
}

```

(a) Test for getSingleSplit.

(b) Test for getBit.

Figure 9: Covering every mutant.

The first couple tests will usually cause line coverage to shot up, although there are some benefits in using line coverage it is not a good enough indication of the tests thoroughness; in many cases, people fall in the tendency of writing tests to archive a high line coverage rather than writing tests that adequately test the software functionality. For example, adding two

new tests was enough to achieve 100% line coverage, in contrast, mutation coverage is way lower at 60% (see figure 10).

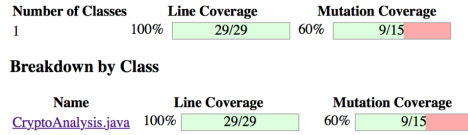


Figure 10: Line Coverage vs Mutation Coverage.

In some cases, it's impossible to create tests that cover every mutant, as would be the case of a mutant generated for some unreachable code; in these cases, you should examine the code carefully and make a decision on whether it should be there at all and if so why is not being used. After the removing any dead code and ensuring that all mutants have been executed, it's time to take a deeper look at the mutants that are still alive (see figure 11 for an update mutation list).

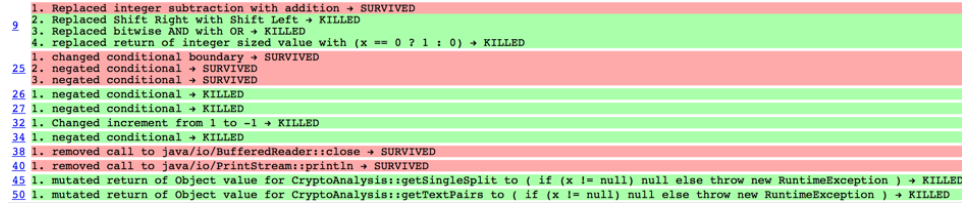


Figure 11: PIT report - Revised Mutations

In order to be able to kill the remaining mutants, one must first understand the mutation and the reason why the test was not able to kill it. Let's examine the code where the following mutation occurred "**Replaced integer subtraction with addition**" (see figure 12 part a), then it's clear that **(31-n)** was mutated to **(31+n)**.

```

static int getBit(int num, int n) {
    return (num >> (31-n)) & 1;
}

```

(a) Source code.

```

assertEquals(crypto.getBit(-1477322454, 0), 1);
assertEquals(crypto.getBit(0, 31), 0);

```

(b) Test for getBit.

Figure 12: Tests.

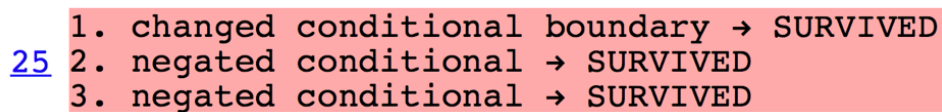
If you recall the asserts from the test created for getBit (see figure 12 part b), my assumption was that testing the first bit (position 0 or leftmost) and

the last (position 31 or right-most) would suffice, I was not totally wrong, in this case, the issues lies in the chosen input as all the bits in zero are zero so getting any of the 32 bits would return the correct result, I know duh...

Many software faults originate from very small mistakes so it makes sense that the same is true for test faults; updating the assert to **assertE-quals(crypto.getBit(1, 31), 1)** killed the mutant. By now it should be becoming more apparent why mutation testing helps designing better tests, typical unit testing aims to find faults in the code, while mutation testing's goal is to find faults in the tests.

4.6 Testing the happy path isn't enough.

Let's examine the next 3 mutants (see figure 13), for most cases deducing the mutation was straightforward, but in cases like this with multiple mutations per line the mutation might not be as apparent; I believe this is where PIT lacks in comparison to its competitors that show the exact mutations.



```
25 1. changed conditional boundary → SURVIVED
    2. negated conditional → SURVIVED
    3. negated conditional → SURVIVED
```

Figure 13: Sad path mutants.

After some digging around, I found out that the test data was faulty; because it was created thinking of the happy path, it had the exact amount of valid input. In order to kill the mutants, I created a new test (see figure 14) and a file with invalid data.

```

@Test
public void test_missing_input() {
    CryptoAnalysis crypto = new CryptoAnalysis();

    try {
        crypto.readKnownTextPairs("/app/src/main/java/with_nulls.txt");

        String [][] pairs = crypto.getTextPairs();

        assertEquals(pairs[0][0], "d3feeda5d5f3d9e4");
        assertEquals(pairs[1][0], "b9a0a168b6b4a416");
        assertNull(pairs[0][1]);
        assertNull(pairs[1][1]);
    }
    catch(IOException e) {
        fail("Did not expect an Exception");
    }
}

```

Figure 14: Test the sad path.

4.7 Mutation coverage makes you rethink your code.

Unfortunately, PIT does not distinguish between the concepts of weakly and strongly killing of mutants, it expects every mutant to be strongly killed, making it impossible to kill some types of mutants without refactoring the code; for example, I had to refactor the code slightly in order to kill the mutant **"removed call to java/io/BufferedReader::close"**, in this case it is ok to declare `BufferedReader` as a public class object making it possible to ensure it was closed during testing (see figure 15 part a).

<pre> try { crypto.bufferedReader.ready(); fail("Expected Buffer to be closed"); } </pre>	<pre> catch(IOException e) { System.out.println("Oh Oh failed to read input file: IOException"); } </pre>
---	---

(a) PIT - requires strongly killing.

(b) Cleaner clode.

Figure 15: PIT - helps you write better code.

Mutation coverage helps detect code that might be irrelevant as is the case of the last mutant in our list **"removed call to java/io/PrintStream::println"**, mutation testing forces you to think about your code rationally, for example, the print function from figure 15 part doesn't actually give the user an extra information that is not already included in the thrown `IOException`, this is an example where sometimes less code might be better.

5 Conclusion

Overall, I found working with PIT very pleasant, it's robust, well documented, efficient and easy to use; with the only negative being that in some cases it can be hard to derive the mutation from the given report, unlike MuJava that explicitly shows each mutation, I believe that is in part related to the fact that this was my first time using the tool and it took some time to get used to it. I was able to achieve 100% mutation coverage as you can see in the final coverage report (see figure 16), it must be said that it was a very long process for such a small portion of code, the team behind PIT have concentrated a lot of effort into efficiency, I would like to see some more being put into developing new features that might help speed up this process.

Number of Classes	Line Coverage	Mutation Coverage
1	100% <div>28/28</div>	100% <div>14/14</div>
Breakdown by Class		
Name	Line Coverage	Mutation Coverage
CryptoAnalysis.java	100% <div>28/28</div>	100% <div>14/14</div>

Figure 16: PIT - Final coverage report

As a software engineer with a couple years experience, I admit that more often than not I write tests that follow the happy path without putting much thought into how will these adapt in the future. I found PIT to be very user-friendly, I particularly liked the fact that its mutation coverage report not only helped me find flaws in some of my tests which I believed to be good, but it also forced me to rethink some of my code which in turn made me understand it better.

	%method	%block	%branch	%line
CryptoAnalysis	100%(7/7)	100%(19/19)	100%(10/10)	100%(28/28)

Figure 17: JCov - Extra coverage stats.

Out of curiosity, I decided to put the code through JCov the coverage tool I used for my first assignment (see figure 17), from the high coverage results, it is safe to assume that using mutation coverage as a criterion for designing unit tests will help ensure a high level of other types of code coverage such as line, node, path, branch, etc.

References

- [1] PIT. [Online]. Available: <http://pitest.org/> (accessed April 1, 2018)
- [2] P. Ammann and J. Offutt. (2008). *Introduction to Software Testing (1 ed.)*, Cambridge University Press, New York, NY, USA.
- [3] Docker. [Online]. Available: <https://www.docker.com/> (accessed April 1, 2018)
- [4] HashiCorp. *Terraform*. [Online]. Available: <https://www.terraform.io/> (accessed April 2, 2018)
- [5] Amazon. *Cloudformation*. [Online]. Available: <https://aws.amazon.com/cloudformation/> (accessed April 2, 2018)
- [6] MuJava. [Online]. Available: <https://cs.gmu.edu/~offutt/mujava/> (accessed April 3, 2018)
- [7] Major. [Online]. Available: <http://mutation-testing.org/> (accessed April 3, 2018)
- [8] H. Coles, T. Laurent, C. Henard, M. Papadakis and A. Ventresque. (2016). *PIT: a practical mutation testing tool for Java (demo)*, Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM New York, NY, USA.
- [9] PIT *Mutators*. [Online]. Available: <http://pitest.org/quickstart/mutators/> (accessed April 5, 2018)
- [10] P. Amman. (2015). *Transforming mutation testing from the technology of the future into the technology of the present*, Google Mutation Workshop 2015. Graz, Austria.