PIT a Practical Mutation Testing Tool for Java (Demo)

Henry Coles
NCR, Edinburgh
henry@pitest.org

Thomas Laurent
Lero@UCD, Ireland & École Centrale de Nantes, France
thomas.laurent@eleves.ec-nantes.fr

Christopher Henard
University of Luxembourg
christopher.henard@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Anthony Ventresque
Lero@UCD, UCD, Ireland
anthony.ventresque@ucd.ie

ABSTRACT

Mutation testing introduces artificial defects to measure the adequacy of testing. In case candidate tests can distinguish the behaviour of mutants from that of the original program, they are considered of good quality – otherwise developers need to design new tests. While, this method has been shown to be effective, industry-scale code challenges its applicability due to the sheer number of mutants and test executions it requires. In this paper we present PIT, a practical mutation testing tool for Java, applicable on real-world codebases. PIT is fast since it operates on bytecode and optimises mutant executions. It is also robust and well integrated with development tools, as it can be invoked through a command line interface, Ant or Maven. PIT is also open source and hence, publicly available at http://pitest.org/

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Mutation Testing, PIT, automated tool

1. INTRODUCTION

Software testing aims at exercising the behaviour of the software, explicit the normal (expected) behaviour of the software and exhibit abnormal behaviour which indicate the presence of bugs (when tests *fail* to run properly). On the contrary, if they *pass*, the program is assumed to have the behaviour expected by the tests. Because of its simplicity and its practicality, software testing has become one of the main software quality assurance techniques in industry. However, adequately measuring the quality of testing is hard. Researchers have proposed several metrics, most of them relying on the notion of *code coverage*, which describes how much of the source code is covered (i.e., merely executed) by the tests. Coverage metrics imply that the more coverage the merrier. This notion, while widely applied, has a major drawback; it only checks if a line/instruction is tested, not how well it is tested.

Mutation testing [4] is a technique that gives a better understanding of what the tests exercise on the program under analysis. Mutation introduces defects, in the form of small code modifications, which should result in an abnormal behaviour when exercised by tests. If the tests fail to expose the defects then the testers/developers can reasonably infer that the tests are not checking every possible behaviour and that the tests need to be improved.

This paper presents PIT, a mutation testing system for Java. PIT is considerably fast as it manipulates bytecode and runs only the tests that have a chances to kill the used mutants (i.e., the tests that execute the instruction where the mutant is located). PIT's major advantage is that it is robust, easy to use and well integrated with development tools [3]. The present paper aims at describing the tool along with its latest improvements on the supported mutants[1]. Previous versions of PIT had a limited set of operators, that we extend to what we call a *extended set* of mutant operators. This set is shown to increase the effectiveness of the mutation process with a limited impact on the execution time [6]. Also, the extended set of operators complies with the standards and the beliefs of mutation testers [1] and thus, making the tool appealing to support future research.

2. MUTATION TESTING

Mutation analysis produces several variants, called *mutants*, of the program under analysis. Mutants are created based on simple syntactic rules, called mutant operators, that transform the syntax of the program, e.g., transform the expression '$a + b$' to a '$a - b$'. Mutants are used to measure how good our tests are by observing and comparing the runtime behaviour of the non-mutated and mutated programs. This is performed based the program output and thus, mutation measures the ability of the tests to project the syntactic program changes to its behaviour, i.e., identifying semantic differences. When mutants exhibit behaviour differences, they are called *killed*, while when they are not, they are called *live*. Mutation testing refers to the process

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1]the new mutants of PIT are still under development and will be released soon. A beta-version of the tool is available at http://hibernia.ucd.ie/PITest++/

of using mutation analysis as a means of quantifying the level of thoroughness of the test process. Thus, it measures the number of mutants that are killed and calculates the ratio of those over the total number of mutants. This ratio represents the adequacy metric and is called *mutation score*.

Mutation has been demonstrated to be quite effective in terms of fault revealing [4] and in mimicking the behaviour of real faults [2]. However, in practice mutation is sensitive to the underlying mutants that it is using. In other words, the set of the realised operators can have a major impact on both scalability [9] and effectiveness [8] of the technique. Therefore, it is mandatory to equip mutation tools with a comprehensive set of mutants that can adequately measure test thoroughness and at the same time is practical. Previous research has proposed to restrict the mutant operators to a small set that we call extended set, e.g., [9] [1], [2] [4], and describe it in the following section.

The most popular Java mutation testing tools are the MuJava [7] and the Major [5]. Unfortunately, these tools were built to support research projects and thus, their practical use is limited [3]. PIT offers the following three advantages: a) it is open source, b) it is well integrated with development tools, as it offers a Maven plugin and c) it is quite robust and actively maintained (operates on the latest version of Java). Details regarding the tools can be found in the work of Delahaye et al. [3].

3. PIT: REAL WORLD MUTATION

PIT is a mutation testing framework for Java developed to support the day to day development on real codebases. This means that PIT aims at:

- having a good integration with build tools (e.g., Maven, Ant, Gradle), integrated development environments (IDEs, such as, Eclipse or IntelliJ) and static code analysis tools (e.g., SonarQube).
- being fast. PIT uses three techniques to obtain its results: working on bytecode instead of source code, selecting the tests to run against the mutants and minimising the number of mutant executions.
- making a clear report of the tests execution. This makes the navigation between source code and mutants easy by highlighting mutants that were not killed.

3.1 Running PIT

PIT is fully integrated to a variety of build tools, IDEs and static code analysis tools. Thus there is no need for additional effort when one of these common tools is used.

To use PIT with Ant or Maven we need to add a task (or plugin) to their build file so that PIT's behaviour is configured. PIT's configuration is straightforward and can be limited to the specific classes we want to test. We can also configure the output directory (for the test reports). A number of other parameters are also offered (e.g., mutation operators, timeout to infer encountered infinite loops).

The extended version of PIT offers an additional option; the generation of a test matrix. This matrix reports for every mutant which tests are killing it. This is particularly important in order to avoid experimental bias due to subsumed [10] and/or duplicated mutants [11].

Once the PIT task has been configured, the build tool can be used without any other concern. This means that PIT will not change the workflow, it will make its process and generate its report without any user interference. The tool will neither leave any artifact nor it will change anything in the compiled code (the mutants are used only by PIT).

3.2 Mutant Generation and Execution

PIT generates mutants via bytecode manipulation. This approach offers significant performance advantages compared to compiling mutant files as it practically reduces the mutant generation cost to zero. Also PIT avoids input output operations and keep memory overhead low. The bytecode representation of the mutants does not require any program to be written on the disk but, instead to keep it in memory (to reduce the memory overheads, only a single mutant is held in memory at a time).

Mutant generation is a two stage process. An initial scan is performed in the main controlling process. All classes in the system under test are examined and possible mutation points (referred to as MutationIdentifiers) are recorded and stored in memory. The mutated bytecode is in fact generated by this scanning process, but is immediately discarded. Only the MutationIdentifiers are stored.

A MutationIdentifier consists of the precise location of the mutation and the name of the mutation operator. The location is specified by the name of the method and class, the method signature and the instruction on which the mutation occurs. This little information is sufficient to recreate each mutant. The descriptions of millions of mutants can therefore be held in memory by the main process.

To asses each mutant by running tests against it, child JVM processes are created. The MutationIdentifier and names of selected tests to run against the mutant are passed to the child by the controlling process. The mutant bytecode is then generated within the child process and inserted into the running JVM using the Java instrumentation API.

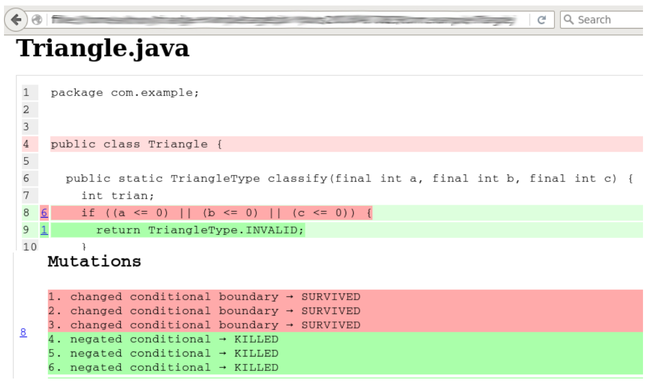
Creating a JVM is a very expensive operation, so PIT tries to minimise the number that are created. Although with a single child, JVM could be used for assess all mutants, the process of running tests against a mutant can leave a JVM in a different state than one that has been freshly started (for example values may be set in static variables or the JVM may become low in memory). This may affect the results of the tests when runed against other mutants. A tradeoff is therefore made between performance and isolation. By default PIT launches a new JVM to asses the mutants related to each class. Hence, it offers a strong guarantee that there will be no interference between mutants in different classes, but does not guarantee that mutants from the same class will not interfere with each other. PIT can be configured to give stronger guarantees (upto and including launching a JVM per mutants) at the expense of performance.

3.3 Extension of the Mutation Operators

In its current release, PIT supports a small number of mutation operators, the objective being to limit the number of mutants and the execution time, but with the risk, as pointed out by some previous studies [1], to have a set of generated mutants of low quality. We have recently proposed [6] to extend the list of mutation operators to increase the effectiveness of the tool. Next Section presents results with respect to both sets of mutants. Table 1 lists all the basic mutation operators (in the current release of PIT) and the ones from the extended list. Note that the current release of PIT has only one mutation operator per relational operator: i.e., PIT mutates $<$ to only $<=$, while there are other possibilities, such as, $< \rightarrow >$, $< \rightarrow =>$, which is taken care of by the extended list of mutation operators.

3.4 Mutation Report

The HTML report generated by PIT uses a colour code to show both the line coverage and the mutation score, see Figure 1 that displays a report on an example application. Light green lines correspond to code coverage with no mutant generated: line 11 in the example. Dark green correspond to the lines for which tests were executed and failed (which means the mutants were killed): this represents the *mutation coverage*, for instance lines 9, 12 and 13 in the example. Light pink shows lines with no code coverage (outside the scope of the tests): line 4 in the example. Dark pink is used to show instructions on which mutants were generated and not detected by the tests (surviving mutants): line 8.



Triangle.java

```
1     package com.example;
2
3
4     public class Triangle {
5
6       public static TriangleType classify(final int a, final int b, final int c) {
7         int trian;
8  6      if ((a <= 0) || (b <= 0) || (c <= 0)) {
9  1        return TriangleType.INVALID;
10        }
```

Mutations

```
   1. changed conditional boundary → SURVIVED
   2. changed conditional boundary → SURVIVED
   3. changed conditional boundary → SURVIVED
8  4. negated conditional → KILLED
   5. negated conditional → KILLED
   6. negated conditional → KILLED
```

Figure 1: Example of output of PIT.

Beside each line number, another number, embedding an internal web link, gives the number of mutants generated for the line. For instance in Figure 1 we can see that there are 6 mutants generated for line 8 (in dark pink). Clicking on the link brings the focus to the list of mutants (at the bottom of the HTML page) where the colour code shows that among the 6 mutants generated for line 8, 3 were killed (dark green) and 3 survived (dark pink).

4. DEMONSTRATION RESULTS

To demonstrate the applicability and the performance of PIT we select 5 Java projects (recorded in Table 3) that are frequently employed in academic evaluations. Table 2 records: the version, lines of code (calculated with the JavaNCSS tool, number of classes (for which test suites exist) and number of tests are reported. Joda-time is a date and time manipulation library. Jfreechart is a popular library for creating charts and plots. Jaxen is an engine for evaluating XPath expressions. Commons-lang provides a set of utility methods for the commons classes of Java. Finally, commons-collections is a set of data structures for Java.

Tables 4 and 5 record the results obtained from the selected programs. These demonstrate that PIT is applicable on real world programs and that the extended operator set is feasible and practical. The results also demonstrate that the results vary when using the two sets indicating the need for using the extended mutants, when possible [6].

Table 3: Test subjects, lines of code (LoC), number of classes and number of test cases.

Subjects	Version	LoC	Classes	Tests
joda-time	2.8.1	18,611	210	4,129
jfreechart	1.0.19	46,986	290	1,320
jaxen	1.1.6	6,790	152	646
commons-lang	3.3.4	16,286	199	3,373
commons-collections	4.4.0	11,281	243	2,210

5. CONCLUSION

This paper presents PIT, a mutation testing tool for Java. PIT is a robust and easy to use mutation testing tool. It is quite popular and has been widely used. In summary, PIT offers the following major advantages: it is well integrated with development tools as it supports both Ant and Maven; it is open source and actively maintained; it is scalable and supports mutant operators that conform to the current practice of mutation testing research.

6. REFERENCES

[1] P. Amman. Transforming mutation testing from the technology of the future into the technology of the present, https://sites.google.com/site/mutationworkshop2015/program/mutationkeynote.pdf?attredirects=0&d=1.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[3] M. Delahaye and L. du Bousquet. Selecting a software engineering tool: lessons learnt from mutation analysis. *Software: Practice and Experience*, 45(7):875–891, 2015.

[4] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[5] R. Just. The major mutation framework: efficient and scalable mutation analysis for java. In *ISSTA*, pages 433–436, 2014.

[6] T. Laurent, A. Ventresque, M. Papadakis, C. Henard, and Y. L. Traon. Assessing and improving the mutation testing practice of PIT. *CoRR*, abs/1601.02351, 2016.

[7] Y. Ma, J. Offutt, and Y. R. Kwon. Mujava: a mutation system for java. In *ICSE*, pages 827–830, 2006.

[8] A. S. Namin and S. Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *ISSTA*, pages 342–352, 2011.

[9] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *TOSEM*, 5(2):99–118, 1996.

[10] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon. Threats to the validity of mutation-based test assessment. In *ISSTA*, 2016.

[11] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *ICSE*, pages 936–946, 2015.

Table 1: PIT's basic mutant operators

Name	Transformation	Example	Name	Transformation	Example
Cond. Bound.	Replaces one relational operator instance with another one (single replacement).	$< \rightsquigarrow \leq$	**Return Values**	Transforms the return value of a function (single replacement).	`return 0` $\rightsquigarrow$ `return 1`
Negate Cond.	Negates one relational operator (single negation).	$== \rightsquigarrow !=$	**Void Meth. Call**	Deletes a call to a void method.	`void m()` $\rightsquigarrow$
Remove Cond.	Replaces a cond. branch with true or false.	`if (...)` $\rightsquigarrow$ `if (true)`	**Meth. Call**	Deletes a call to a non-void method.	`int m()` $\rightsquigarrow$
Math	Replaces a numerical op. by another one (single replacement).	$+ \rightsquigarrow -$	**Con-structor Call**	Replaces a call to a constructor by null.	`new C()` $\rightsquigarrow$ `null`
Incre-ments	Replace incr. with decr. and vice versa (single replacement).	$++ \rightsquigarrow --$	**Member Variable**	Replaces an assignment to a variable with the Java default values.	`a = 5` $\rightsquigarrow$ `a`
Invert Neg.	Removes the negative from a variable.	$-a \rightsquigarrow a$	**Switch**	Replaces switch statement labels by the Java default ones.	
Inline Const.	Replaces a constant by another one or increments it.	$1 \rightsquigarrow 0$, $a \rightsquigarrow a+1$			

Table 2: Extended mutant operator list of PIT

Name	Transformation	Example	Name	Transformation	Example
ABS	Replaces a variable by its negation.	$a \rightsquigarrow -a$	**OBBN**	Replaces the operators & by | and vice versa.	$a\&b \rightsquigarrow a\|b$
AOD	Replaces an arithmetic expression by one of the operand.	$a+b \rightsquigarrow a$	**ROR**	Replaces the relational operators with another one. It applies every replacement.	$< \rightsquigarrow \geq$, $< \rightsquigarrow \leq$
AOR	Replaces an artihmetic expression by another one.	$a+b \rightsquigarrow a*b$	**UOI**	Replaces a variable with a unary operator or removes an instance of an unary operator.	$a \rightsquigarrow a{+}{+}$
CRCR	Replaces a constant a with its negation, or with 1, 0, $a+1$, $a-1$.	$a \rightsquigarrow -a$, $a \rightsquigarrow a-1$.	**Com-mons**	*All the common operators as described above.*	

Table 4: Number of mutants, killable and mutation score (MS) for the basic and extended lists of operators.

Measure		joda-time Basic	joda-time Extended	jfreechart Basic	jfreechart Extended	jaxen Basic	jaxen Extended	commons-lang Basic	commons-lang Extended	commons-collections Basic	commons-collections Extended
#Mutants	*Min.*	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Med.	97.00	224.00	98.00	260.50	24.00	39.00	27.00	57.00	27.00	42.00
	Mean	164.17	462.06	219.14	685.48	77.48	188.97	156.82	457.05	62.32	126.53
	Max.	973.00	2,915.00	3,436.00	9,742.00	3,901.00	14,493.00	4,545.00	14,586.00	1,094.00	2,349.00
#Killable	*Min.*	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Med.	60.99	136.99	26.00	49.00	11.99	21.00	17.00	33.50	5.00	5.00
	Mean	117.32	295.71	59.59	131.20	37.91	69.31	124.74	338.86	21.66	41.34
	Max.	834.00	2,108.00	1,356.00	2,488.00	773.00	1,793.00	3,928.99	11,522.99	867.00	1,553.00
MS	*Min.*	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Med.	0.80	0.71	0.24	0.16	0.73	0.66	0.84	0.74	0.50	0.45
	Mean	0.71	0.64	0.29	0.24	0.60	0.56	0.72	0.66	0.44	0.41
	Max.	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 5: Execution time in seconds for the basic and extended lists of operators.

	Basic			Extended		
Subjects	Mutants	Killable	Time	Mutants	Killable	Time / Overhead
joda-time 2.8.1	35,297	25,224	1,138	99,343	63,578	3,531 / 210%
jfreechart 1.0.19	81,960	22,289	2,398	256,370	49,069	6,589 / 175%
jaxen 1.1.6	14,334	7,014	1,221	34,960	12,823	31,077 / 2,445%
commons-lang 3 3.4	34,502	27,443	2,803	100,553	74,550	8,023 / 186%
commons-collections 4 4.0	24,308	8,449	570	49,354	16,126	1,230 / 116%