JOIV : Int. J. Inform. Visualization, 6(2-2): A New Frontier in Informatics - August 2022 455-462

INTERNATIONAL JOURNAL ON INFORMATICS VISUALIZATION

journal homepage : www.joiv.org/index.php/joiv

The Investigation of Java Mutation Testing Tools

Sara Tarek ElSayed Abbas [a,b], Rohayanti Hassan [a,b,*], Shahliza Abd Halim [a,b], Shahreen Kasim [c], Rohaizan Ramlan [d]

[a] *Computer Science Department, AL-Salam University College, Hay AL-khadra'a, Baghdad, 10022, Iraq*
[b] *School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, 81310 Johor Bharu, Johor, Malaysia*
[c] *Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia, 86400 Batu Pahat, Johor, Malaysia*
[d] *Production and Operation Management Department, Faculty of Technology Management and Business, Universiti Tun Hussein Onn Malaysia, 86400 Batu Pahat, Johor, Malaysia.*
*Corresponding author: *rohayanti@utm.my*

***Abstract*—Software Testing is one of the most significant phases within the software development life cycle since software bugs can be costly and traumatic. However, the traditional software testing process is not enough on its own as some undiscovered faults might still exist due to the test cases' inability to detect all underlying faults. Amidst the various proposed techniques of test suites' efficiency detection comes mutation testing, one of the most effective approaches as declared by many researchers. Nevertheless, there is not enough research on how well the mutation testing tools adhere to the theory of mutation or how well their mutation operators are performing the tasks they were developed for. This research paper presents an investigative study on two different mutation testing tools for Java programming language: PIT and µJava. The study aims to point out the weaknesses and strengths of each tool involved through performing mutation testing on four different open-source Java programs to identify the best mutation tool among them. The study aims to further identify and compare the mutation operators of each tool by calculating the mutation score. That is, the operators' performance is evaluated with the mutation score, with the presumption that the more prominent the number of killed mutants is, the higher the mutation score, thus the more effective the mutation operator and the affiliated tool.**

***Keywords*— Mutation testing; mutation score; PIT tool; µJava tool, JUnit.**

Manuscript received 17 Dec. 2021; revised 29 Jan. 2022; accepted 21 Apr. 2022. Date of publication 31 Aug. 2022.

I. INTRODUCTION

Software testing is the process within the SDLC that plans, prepares, and evaluates the features of a software item, intending to point out the underlying defects to ensure that the Software Under Test (SUT) meets the identified software requirements [1]. The process has various purposes, including the prevention of software failure, running tests to find failures, measuring quality, and providing confidence in the product [2]. The technique is now a key component of software product development because reliability, security, and high performance are guaranteed from a properly tested software product which further leads to saving time, money and ensuring customer satisfaction.

White-box testing is a technique that aims at testing the internal structure of a software system, in contrast to its counterpart black-box testing that instead evaluates the functionality without looking neither at the software's internal structure nor operations [3]. Through white-box testing, the internal components get exercised to achieve the most satisfactory outcome. The typical white box testing process involves the scenario of test cases generation to test the inner workings of an application. The quality gets assessed via the created test cases by exercising the paths through the code and comparing the anticipated results with the actual outputs. However, this is not enough on its own because some undiscovered faults might still exist due to the test cases' inability to detect all underlying faults. For that reason, researchers developed various techniques to detect if the test cases are good at revealing the underlying faults and capable of properly evaluating the quality of a test suite. Amidst the numerous techniques proposed comes a fault-based testing strategy known as Mutation Testing.

Mutation Testing is a fault-injection white-box testing technique where a few statements of the source code are deliberately mutated with mutation operators to ensure that the test cases are capable of detecting errors in the software. For instance, adding mutation operators to replace arithmetic

operations, alter increments, change logical operators, remove a line of code, or set an assignment to a hard-coded value instead of a variable. In case the tests detect a different test result for the mutant than the one it has for the original version of the program, then we can conclude that the test case is effective since it has killed the generated mutant by detecting the syntactic error. In this respect, the mutant is said to be dead as the inserted error was identified successfully by the test cases. On the other hand, if the code runs successfully with no recognized errors, then the test case is deemed inadequate due to its inability to detect the presence of errors, and the mutant, in this case, is said to be live.

The traditional mutation testing process commences with the original program P, where a few faults are introduced by a set of mutators m, generating a collection of mutated programs P' known as mutants. For the generated mutants, a collection of test cases is executed, and the behavior of the tests is monitored and assessed accordingly to denote their efficiency. An example of the mutants' formulation is shown in **Error! Reference source not found.**. The mutated program P' is produced via mutator m, which switches the or operator (||) of the original program into the and operator (&&), thus forming the mutant version of the program P'.

TABLE I
APPLYING A MUTATION OPERATOR TO A PROGRAM

Original Program P	**Mutant P'**
if (int i > 0 || int q > 0)	if (int i > 0 && int q > 0)
return 1;	return 1;

The mutation testing technique is considered one of the most adequate approaches to assess the quality of a test suite in several domains [4]. It is effective to the point of subsuming almost all other structural testing techniques [5]. However, despite the effectiveness of mutation testing in evaluating the quality of test suites, it still suffers from several problems that shall be addressed.

Mutation testing has failed to gain widespread adoption in the software engineering practice, and make its industrial debut [6]–[9]. These limitations led to the dismissal of the technique by many software testers and organizations despite its effectiveness. Researchers reveal the unpopularity of industrial mutation testing is due to the hefty expenses associated with the procedure [10]–[12]. Generating, running, and executing an enormous number of mutants against a test set is considered very expensive [13], time-consuming, and onerous as it requires substantial computational resources that call for large storage space. Wherefore, it is significant to save time, effort [14], and resources by using an automated, fast, and reliable mutation testing tool. Software testing, however, is a costly process, costing more than 50% of the whole development expenses. This is why decreasing the costs of the technique and improving the test efficiency through automating the process of software testing is significant [15].

Another issue affiliated with the technique is that the performance of mutation testing varies from one tool to another. Whereas different results could be obtained for the same test suites owing to the fact that the strength of mutation testing and its effectiveness highly depends on the set of mutants used in the mutation process. In other words, different sets of mutants that are based on well-defined mutation operators can lead to different results [16]–[18]. Therefore, it is necessary to identify and use the most optimal set of mutation operators that generates an effective collection of mutants, efficiently assessing the adequacy of the test suites.

There is a diversity of tools for mutation testing, each with its different applications, restrictions, and set of used mutants. The different implementations and limitations of the tools can lead to different results in terms of the effectiveness of the test suites [19]–[21]. Furthermore, since the technique is very time-consuming, it is necessary to use a reliable, fast, and automated mutation testing tool. The tool shall be efficient in terms of its performance and capability of generating mutants that are to be run and executed against a suite of tests [22], [23]. The tool shall further be effective in reporting the mutation score.

This paper presents an investigative study of two distinctive Java Mutation Testing tools: μJava and PIT. The study aims to perform mutation testing on four different open-source Java applications to identify the best mutation tool among them. The study aims to further identify and compare the mutation operators of each tool in terms of fault detection rate through calculating the mutation score. This is done with the presumption that the more prominent the number of killed mutants is, the higher the mutation score, thus the more effective the mutation operator and the affiliated tool.

II. MATERIAL AND METHOD

This investigative study on Java Mutation Testing Tools is briefly about conducting a mutation analysis on different applications written using Java programming language with the help of two mutation testing tools: μJava and PIT, to determine which tool is more efficient and effective in detecting the injected mutants. This will be achieved by comparing the number of killed mutants to the total number of mutants generated to calculate the overall mutation score for each test suite. This section presents the experimental setup carried out for mutation testing process.

A. Experiment and Evaluation

Firstly, the environment for mutation testing is setting up, which includes installing the appropriate tools used in the study. Afterwards, the experiment will start operating where the process of mutation testing will be implemented on the selected Java applications to test the performance of the mutation operators for each one of the two tools used in the study. For evaluation, mutation score metric will be used to assess the performance of the tools with their respective operators. Mutation score is a fault-based technique that measures how well a set of operators are performing. Eq. 1 displays the Formula of Mutation Score. *'P* shows the generated mutants, *P* represents the original program under test, *T* represents the test suite, *K* is the total number of killed mutants and *E* displays the equivalent mutants. The Mutation testing score is the percentage of the total killed mutants to the total number of generated mutants (excluding equivalent mutants in the program).

$$\boldsymbol{MS}(P,\boldsymbol{T}) = \frac{K}{(\prime P - E)} \quad (1)$$

B. μJava Tool

μJava—*Mutation System for Java*, is an open-source automated mutation testing tool innovated by the Korea Advanced Institute of Science and Technology (KAIST) in South Korea and George Mason University in the USA's collaboration. The tool allows the testers to perform mutation analysis on Java programs by automatically generating a set of mutants based on the mutation operators selected. Then, it further executes the supplied JUnit test cases against the mutants produced, reporting the mutation testing score accordingly. As illustrated in **Error! Reference source not found.**, μJava tool contains three main components*: 1) mutants' generator*, 2) *mutants' viewer,* and 3) *mutants' executor*.

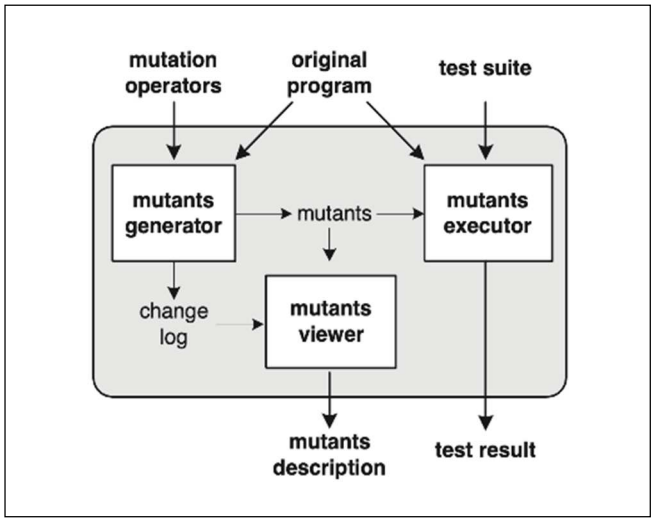

Fig. 1 μJava's Structural Architecture

The *mutant generator* is responsible for generating the mutants, which could be traditional mutants or class mutants. The mutants generated are produced in the form of source code, which is then compiled into byte code. Accordingly, the details of every single mutant could be observed from the *mutants' viewer* that displays the number and type of generated mutants, along with the manipulated part of the code. The *mutants' executor* executes the mutants against the test suite and accordingly calculates its test score.

C. PIT Tool

Parallel Isolated Test (PIT)—alternatively known as Pitest, is one of the best performing Mutation Testing tools for java that is well known for its fast process of mutants' generation [24]. The tool [25]–[27] is designed to conduct mutation analysis on java programs to support the software testers with the process of mutation testing on real-world codebases. It can be used via the Command Line Tool, as a plugin in Maven, Eclipse, IntelliJ IDEA, and Gradle, or as an Ant task. In this study, IntelliJ's *PIT mutation testing Idea* plugin version 1.4.5 by Michael Jedynak is used.

There are four different phases of mutation testing in PIT. The first phase is the *Mutants Generation* phase that generates mutants in PIT by manipulating the bytecode of the compiled classes using the assistance of java's ASM library. The second phase, *Test Selection,* is the phase where code coverage is used to select only the tests executing the link, block, or instruction containing the mutant that will be run against. Measuring the code coverage in PIT allows only the tests that could kill the mutant to run. The third phase is *Mutant Insertion.* In this phase, PIT inserts the mutants via JVM using the instrumentation API. The generated mutants are all held in memory without getting stored locally. The fourth and last phase is the *Detection of Mutants* where the test classes are divided into individual test cases that are run accordingly until one of these test cases kills a mutant.

D. Subject Programs

To test the selected tools, μJava and PIT, four Java programs were selected of different lengths and complexity. **Error! Reference source not found.** describes the applications used in the experiment.

TABLE II
PROGRAMS USED IN THE EXPERIMENT

Java Program	Classes	LOC
Min	Min	27
Calculator	Calculator	19
TriTyp	TriTyp	68
CoreBanking	account.account	252
	corebanking	107
	datamanagement.ReadFromDB	63
	transaction.BankTransaction	96
	transaction.CashTransaction	101

E. Experimental Design

After conducting an extensive study on the mutation testing domain, a simple flowchart as illustrated in Fig. explains the experimental design to-be followed for this research. To start the processes of Unit and Mutation Testing, the environment needs to be suitable for the software testing process with Java. Thus, the first step is the installation and setup of the Java environment (installing JDK 8), followed by an installation of JUnit 4 for the process of unit testing.

Thereafter, the subject programs (the Java program files used in this study, namely Min, Calculator, TriTyp, and CoreBanking) shall be installed into the machine. Once the files are installed successfully, they shall be loaded into IntelliJ IDEA for the purpose of Unit Testing. For each java program used in this study, a set of JUnit 4 test cases is written. All the actual results obtained from the tests shall match the expected results once executed by having a status of pass. In case the tests have a failed status, then they shall be revised until they all pass successfully.

To ensure that the tests created are adequate, we shall not only rely on the tests' *pass/fail* status. Thus, all tests are further tested via test coverage. The required test coverage criteria are 100% line and method coverage. For all tests, if one failed to achieve either 100% line or method coverage, it is fixed until the coverage metrics are successfully met. Once this step is completed, then the succeeding step of setting up the environment for PIT and μJava begins. The download of μJava requires an installation of mujava.jar, openjava.jar and mujava.config files. Once downloaded, the CLASSPATH is changed, the mujava.config file is set to point to μJava's directory, and the subdirectories that shall contain the source files, class files, test sets and the results get created.

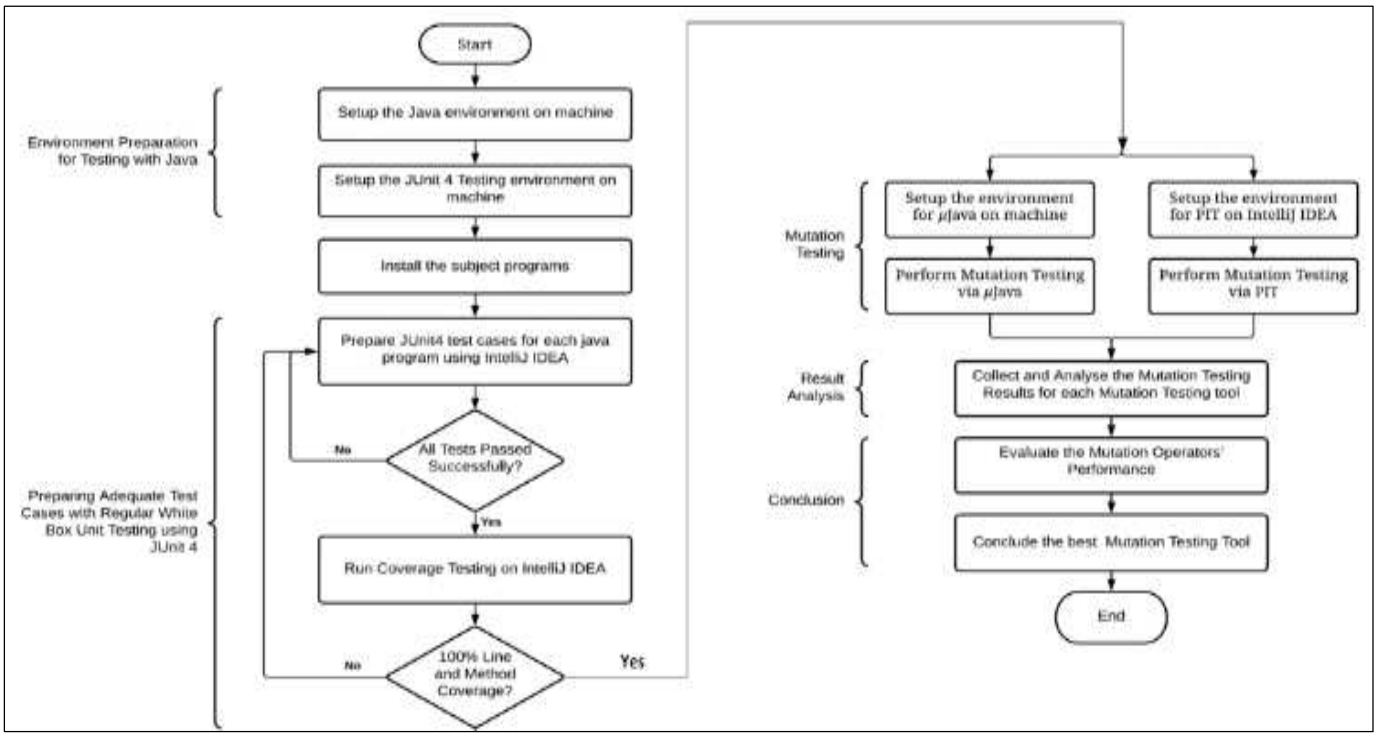

Fig.1 Experimental Design Flowchart

The download of PIT on the other hand is performed using Maven plugin in IntelliJ IDEA. The plugin is added to the build file in pom.xml to configure the tool. After the successful setup of the environment, mutation testing begins. The process is performed on each tool independently by generating mutants via the mutation operators and the mutants are then executed against the created tests. The results obtained are analyzed to assist with the evaluation process. Once the mutation testing process completes and all results are analyzed, the obtained findings shall indicate the performance of the mutation tools and their affiliated operators, marking the end of the research.

F. White-box Unit Testing

This section covers the procedure and implementation of mutation testing with PIT and μJava to evaluate their overall performance along with detecting their best-performing mutation operators. Regular white-box automation testing is performed as a start on the four subject programs via JUnit 4 testing framework. Test coverage is further performed on the developed tests to ensure that they are adequate for usage in the mutation testing process. Once the coverage metrics are met, the actual mutation testing process begins.

The mutation testing is performed independently via each tool to point out the defects that were not detected beforehand during the regular white-box testing. In mutation testing, the mutants shall be generated for each subject program once via PIT's mutation operators and once via μJava's mutation operators. Next, these mutants shall be executed accordingly on the generated test cases. The results obtained from the mutation testing process are then gathered and further analyzed to achieve the objectives of this study.

Initially, each subject program is loaded individually into the *src* directory in JetBrains' IntelliJ IDEA. Then, a new directory is added to the project structure to hold the test files created for testing the SUT. It is crucial to maintain the project's structure by marking the directories either as *Sources* (sources root) or *Tests* (test sources root) from the Project Settings as shown in Fig.; otherwise, the IDE will not be able to recognize nor differentiate the test files from the source files and an issue would arise as a consequence.

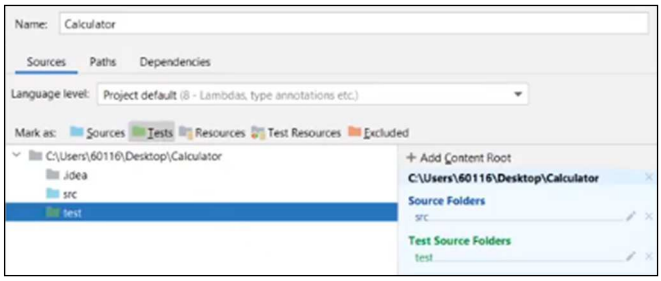

Fig.3 Marking the Project Directories in IntelliJ IDEA

Once the project structure is maintained, JUnit can lastly be added to the classpath via Maven (the JUnit 4.12 library will be downloaded from Maven Repository) for the IDE to recognize the JUnit test cases that will be created later on in the software testing process. Now that the test environment is configured currently in IntelliJ IDEA, the process of Unit testing starts.

The unit testing process begins with the creation of unit tests, which are small programs created to exercise and execute the SUT under defined conditions and with specified inputs. Upon the successful creation of test cases, they are executed via the test runner to exercise the SUT.

As an outcome of a successful test run, the test results are produced. As displayed in Eq. 2, for a test t running on program P, a test result r that is determined by the test oracle is produced. The result can either be a *pass* or a *fail* for a given test.

$$T(P) = r, where\ r \in \{pass, fail\} \quad (2)$$

G. Mutation Testing

Traditional coverage metrics that are used to evaluate the quality of unit-level tests measure how effectively the test inputs exercise certain code structures. However, these metrics have no way of judging the quality of the checks used in detecting defects. That is why mutation testing is an important concept, as the technique, unlike the traditional code coverage, assesses and improves the quality of software tests not only in terms of coverage, but also in terms of checks.

The mutation testing process is performed via mutation testing tools, each with its own unique set of operators. These operators are used to introduce defects into the original program by generating mutants. The mutants aim at mimicking the typical errors the developers are prone to making. In other words, a mutant is a result of applying a mutation operator into a given program. Mutant detection as perceived from Eq. 3 is when a mutant *'P* is detected by a test suite *T* (containing test *t*) due to the different result produced for the original program *P* than for the mutant.

$$\exists_{t \in T}('P) \neq t(P) \quad (3)$$

1) Mutation Testing with PIT: The first step of the mutation testing process with PIT is preparing the mutation testing environment for PIT on IntelliJ IDEA by installing IntelliJ's PIT mutation testing Idea plugin. Right after, the workspace shall be prepared by loading the project file(s) into the IDE, followed by loading the test files, then adjusting the project structure accordingly. These steps are similar to the ones carried out previously at the beginning of the white box testing process with IntelliJ.

Given that the environment on IntelliJ is ready for mutation testing, the actual process of mutation testing with PIT begins. The operation starts by choosing *Pitest all tests in module* option attained by right clicking the project folder from the project viewer in IntelliJ. As an outcome, PIT Run Configuration window appears.

The window displays the information related to the mutation testing process performed. This information includes the result of mutation testing for each operator used in the testing process, the number of mutants each operator has generated, the number of mutants killed by the test suite, and the number of mutants that have survived and the mutation score for this given operator. Moreover, at the end of the Run Configuration window lies a *Statistics* section summing up the total result of the mutation testing performed.

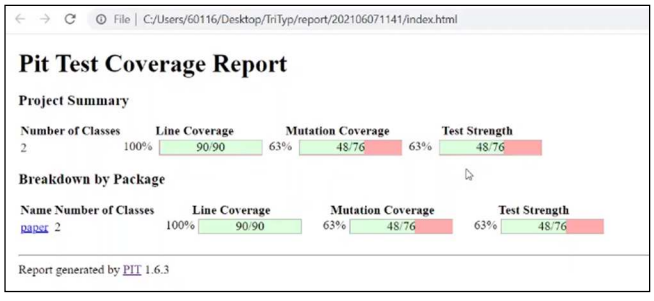

Fig. 4 Pit Test Coverage Report for TriTyp Project

A feature of PIT is the HTML report generated by the end of the testing. This report contains a summary of the mutation testing performed. To open the PIT Test Coverage Report, the hyperlink titled *Open report in browser* lying by the end of the PIT Run Configuration Window is clicked. An example of the Pit Test Coverage Report is displayed in Fig. .

2) Mutation Testing with µJava: To perform mutation testing with µJava, a few steps are ought to be followed. First, the source file(s) of the subject program(s) shall get added into the *src* folder found in *µ*Java's main directory. Then, the compiled versions of these source files are to be added to the *classes* folder. Lastly, the compiled versions of the test scripts are to be added to the *testset* folder.

The next step is to generate mutants. This target is achieved by launching the Mutants Generation GUI via the execution of the command java mujava.gui.GenMutantsMain. From the Mutants Generator GUI, the source files that will be mutated are selected from the Files section. Then, the mutation operators that will be applied to these source files are chosen from the list of mutation operators provided by the tool. For the mutators to generate the mutants, the Generate button is pressed. Assuming that all of the steps are completed successfully, the generated mutants are added by *µ*Java into the result folder.

Lastly, the tests are executed against the generated mutants. To do this, the command: java mujava.gui.RunTestMain is executed from the terminal to launch µJava's Test Case Runner GUI. From the GUI, the class file to be tested is selected, followed by the selection of the methods, and the test files related to the chosen class file. Then, the run button is hit for it to run tests. The number of live, killed, and total mutants along with the Mutation Score for the Traditional and Class Mutants shall be displayed as a result. Fig. 1 is an example of this step once applied to TriTyp program.

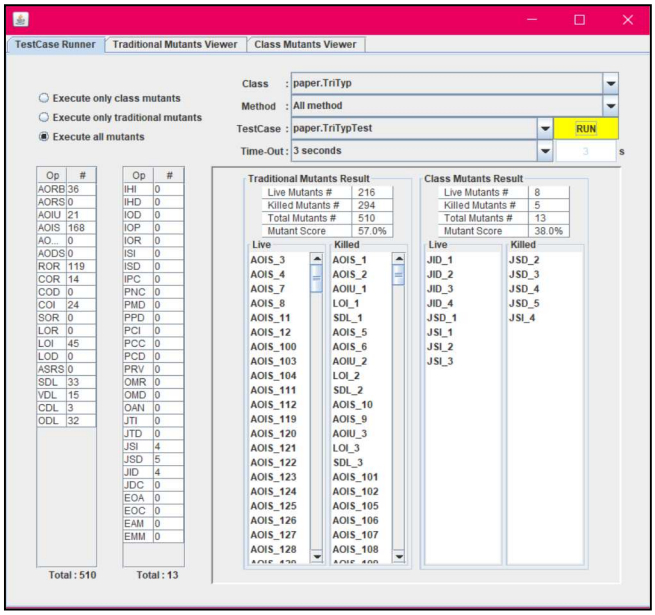

Fig. 1 µJava's Test Case Runner GUI

III. Results and Discussions

This section displays and analyzes the results and observations obtained during the mutation testing process

performed with PIT and μJava. The result analysis is used in the evaluation of the two tools and their operators' performance to conclude the best performing mutation testing tool among them.

A. PIT Results

Error! Reference source not found. gathers the overall number of Mutation Operators applied via PIT on the four subject programs. The table shows the total number of generated, killed, and live mutants for each mutator in PIT, in addition to the mutation score. Overall, *Void Method Call Mutator* was applied the most throughout the mutation testing process, followed by *Negate Conditionals Mutator*. The total number of generated mutants for *Boolean True Return Vals Mutator, Null Return Vals Mutator, Boolean False Return Vals Mutator,* and *Primitive Returns Mutator* stood under ten mutants. The tests managed to kill all mutants produced by *Boolean True Return Vals* and *Null Return Vals* Mutators, followed by *Empty Return Vals Mutator* and *Negate Conditionals Mutator* with a total mutation score of 91%, and lastly at the third place comes *Math Mutator* with 81% Mutation score. To detect the best performing mutants for PIT, Boolean False Return Vals, Boolean True Return Vals, Null Return Vals and Primitive Returns mutators were eliminated from the comparison because they have generated a fewer number of mutants than the rest of the operators. As for the performance of PIT on the subject programs, a summary of the overall mutation testing result for each program can be observed from 4.

TABLE III
OVERALL RESULTS FOR PIT MUTATION TESTING OPERATORS

PIT Mutation Operator	Total Generated Mutants	Total Live Mutants	Total Killed Mutants	Mutation Score
Boolean True Return Vals Mutator	3	0	3	100%
Empty Object Return Vals Mutator	11	1	10	91%
Conditionals Boundary Mutator	15	11	4	27%
Void Method Call Mutator	83	19	64	77%
Null Return Vals Mutator	8	0	8	100%
Math Mutator	16	3	13	81%
Boolean False Return Vals Mutator	1	1	0	0%
Negate Conditionals Mutator	45	4	41	91%
Primitive Returns Mutator	7	2	5	71%

TABLE IV
PIT OVERALL MUTATION RESULT

Java Program	Classes	Mutant Generated	Live Mutant	Killed Mutant	Mutation Score
Min	Min	13	8	5	38%
Calculator	Calculator	9	4	5	56%
TriTyp	TriTyp	76	28	48	63%
CoreBanking	account.account	26	3	23	88%
	corebanking	11	7	4	36%
	datamanagement. ReadFromDB	17	7	10	59%
	transaction.Bank Transaction	21	4	17	81%
	transaction.Cash Transaction	16	5	11	69%
Total		189	66	123	65%

B. μJava Results

The overall number of mutation operators executed against the subject programs via μJava and their performance in terms of the total number of generated, live, and killed mutants are gathered in **Error! Reference source not found.**. For all mutation operators applied in the study, *AOIS* that generated 262 mutants comes in first place in terms of the highest number of mutants generated, followed by *the ROR* mutator with a total of 213 mutants. *SDL* operator comes in third place with 176 mutants, and *ODL* with 131 mutants in fourth place. The mutators generating the least number of mutants are *IHI*, JTI, and *COD* (only one mutant is generated for each operator). All mutators generating a total number of mutants below ten got eliminated from the comparison since their mutation generation value is low compared to the rest of the mutators. After the elimination, there is a total number of 10 mutators. μJava's performance against the subject programs used in this study is summarized in **Error! Reference source not found.**6. For each program, the table shows the total number of generated, surviving, and killed mutants, in addition to the mutation score.

TABLE V
OVERALL RESULTS FOR μJAVA MUTATION TESTING OPERATOR

PIT Mutation Operator	Total Generated Mutants	Total Live Mutants	Total Killed Mutants	Mutation Score
AORB	64	20	44	69%
AORS	4	2	2	50%
AOIU	47	15	32	68%
AOIS	262	163	99	38%
AODU	1	1	0	0%
ROR	213	73	140	66%
COR	16	9	7	44%
COD	1	0	1	100%
COI	56	7	49	88%
LOI	74	18	56	76%
ASRS	8	1	7	88%
SDL	176	55	121	69%
VDL	31	18	13	42%
CDL	30	7	23	77%
ODL	131	44	87	66%
IHI	1	0	0	0%
IOD	3	0	3	100%
PRV	3	1	2	67%
JTI	1	0	1	100%
JSI	9	6	3	33%
JSD	6	2	4	67%
JID	4	4	0	0%
EAM	7	2	5	71%

TABLE VI
μJAVA OVERALL MUTATION RESULT

Java Program	Classes	Mutant Generated	Live Mutant	Killed Mutant	Mutation Score
Min	Min	72	26	46	63%
Calculator	Calculator	28	3	25	89%
TriTyp	TriTyp	523	224	299	57%
CoreBanking	account.account	327	122	205	62%
	corebanking	15	9	6	40%
	datamanagement. ReadFromDB	32	10	22	69%
	transaction.Bank Transaction	59	12	47	80%

	transaction.CashTransaction	96	47	49	51%
Total		189	1152	453	699

C. Overall Results

The different test suites, adequate for PIT's operators, were also measured in their ability to detect the mutants generated by μJava. The results obtained from the process are illustrated in Fig. 2, which shows the mutation score for each subject program via μJava and PIT. For some programs, μJava has managed to kill a higher number of mutants than PIT, which led to better results for the given program. For instance, μJava has produced a higher mutation score in Calculator, Min, and BankTransaction with 63%, 89%, and 69% mutation scores, respectively, in contrast to PIT with 38%, 56%, and 59% consequently. Nevertheless, PIT has had a better performance for other programs such as TriTyp, Account, and CoreBanking with 63%, 88%, and 69% each compared to μJava's performance of 57%, 62%, and 51%.

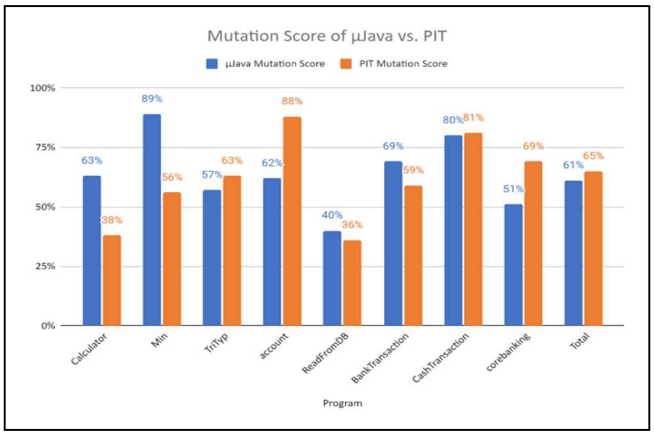

Fig. 2 A comparison between PIT and μJava's Mutation Score for all subject programs

Though the mutation score varies from one program to another for each tool, the overall performance of PIT exceeded μJava's with a mutation score of 65%. This leads to the conclusion that PIT is the best performing tool. Aside from the mutation score, the number of mutants generated in addition to the tools' timings play an important role when it comes to assessing the tools' effectiveness. As Fig. 3 suggests, μJava produces far more mutants than PIT. For instance, μJava produced a number of mutants that is seven times as much as PIT's in *TriTyp,* and for CoreBanking's case, it generated a sextuple of PIT's number of mutants.

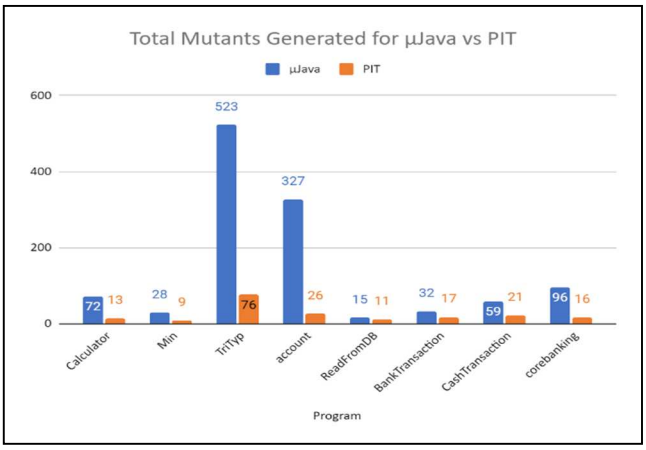

Fig. 3 A comparison between PIT and μJava's total number of generated mutants all subject programs

This suggests that a large number of μJava's mutation operators are redundant and leads to the conclusion that PIT employs an efficient set of operators that values quality over quantity, which strengthens the claim that PIT is the best performing mutation testing tool for java.

IV. Conclusion

This research carried out a comparative experiment on two different Java mutation testing tools: μJava and PIT, which used the assistance of the four subject programs: Min, Calculator, TriTyp and CoreBanking. The process of mutation testing was performed to compare the behaviour of the two tools, leading to the identification of the most efficient tool among them based on the results concluded from the experiment: PIT (with a total overall mutation score of 65% in comparison to μJava's 61%). Moreover, the best performing mutation operators were further evaluated in terms of their effectiveness. The objective was achieved with the help of the mutation score (ms), which calculates the performance of the test suites dividing the percentage of the number of killed mutants by the test case over the total number of generated mutants. As a result, the best performing mutation operators for PIT were Empty Object Return Vals with 91% ms, Negate Conditionals with 91% ms and Math with 81% ms. The best performing μJava mutation operators on the other hand were COI with 88% ms, CDL with 77% ms and LOI with 76% ms.

Acknowledgment

This work was supported/funded by the Ministry of Higher Education under Fundamental Research Grant Scheme (FRGS/1/2020/ICT02/UTM/03/1).

References

[1] P. Mudholkar, M. Mudholkar, and S. Kulkarni, "Software testing," Proc. Int. Conf. Work. Emerg. Trends Technol., pp. 1024–1024, Feb. 2010, doi: 10.1145/1741906.1742242.

[2] R. Black, E. van Veenendaal, and D. Graham, Foundations of software testing : ISTQB certification. 2012.

[3] S. Sangwan, "Software Testing Techniques and Strategies," Isha Int. J. Eng. Res. Appl., vol. 4, no. 4, pp. 99–102, 2014.

[4] P. Ammann and J. Offutt, Introduction to Software Testing. 2007.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," Proc. - 27th Int. Conf. Softw. Eng. ICSE05, pp. 402–411, 2005, doi: 10.1145/1062455.1062530.

[6] B. Falah, M. Akour, and S. Bouriat, “RSM: Reducing mutation testing cost using random selective mutation technique,” Malaysian J. Comput. Sci., vol. 28, no. 4, pp. 338–347, 2015, doi: 10.22452/MJCS.VOL28NO4.5.

[7] S. Hamimoune and B. Falah, “Mutation testing techniques: A comparative study,” Nov. 2016, doi: 10.1109/ICEMIS.2016.7745368.

[8] M. Al-Hajjaji, J. Krüger, F. Benduhn, T. Leich, and G. Saake, “Efficient Mutation Testing in Configurable Systems,” in Proceedings - 2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design, VACE 2017, 2017, pp. 2–8, doi: 10.1109/VACE.2017.3.

[9] P. Jung, S. Kang, and J. Lee, “Efficient regression testing of software product lines by reducing redundant test executions,” Appl. Sci., vol. 10, no. 23, pp. 1–21, Dec. 2020, doi: 10.3390/app10238686.

[10] M. Hafiz, “Mutation Testing Tool for Java,” 2008.

[11] M. Delamaro and J. C. Maldonado, “Proteum - A Tool for the Assessment of Test Adequacy for C Programs User’s guide,” 1996.

[12] D. Singh and B. Suri, “Mutation testing tools-An empirical study,” IET Conf. Publ., vol. 2013, no. CP646, pp. 230–239, 2013, doi: 10.1049/CP.2013.2596.

[13] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” IEEE Transactions on Software Engineering, vol. 37, no. 5. pp. 649–678, 2011, doi: 10.1109/TSE.2010.62.

[14] M. S.GeethaDevasena and M. L. Valarmathi, “Search based Software Testing Technique for Structural Test Case Generation,” Int. J. Appl. Inf. Syst., vol. 1, no. 6, pp. 20–25, 2012, doi: 10.5120/ijais12-450185.

[15] S. Anand et al., “An orchestrated survey of methodologies for automated software test case generation,” J. Syst. Softw., vol. 86, no. 8, pp. 1978–2001, Aug. 2013, doi: 10.1016/j.jss.2013.02.061.

[16] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, “Threats to the validity of mutation-based test assessment,” in ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis, Jul. 2016, pp. 354–365, doi: 10.1145/2931037.2931040.

[17] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, “Does choice of mutation tool matter?,” Softw. Qual. J., vol. 25, no. 3, pp. 871–920, 2017, doi: 10.1007/s11219-016-9317-7.

[18] M. Delahaye and L. Du Bousquet, “Selecting a software engineering tool: Lessons learnt from mutation analysis,” in Software - Practice and Experience, Jul. 2015, vol. 45, no. 7, pp. 875–891, doi: 10.1002/spe.2312.

[19] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, “Analysing and comparing the effectiveness of mutation testing tools: A manual study,” in Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Dec. 2016, pp. 147–156, doi: 10.1109/SCAM.2016.28.

[20] B. Venners, “Test-Driven Development,” in A Conversation with Martin Fowler, Part V, 2002, p. http://www.artima.com/intv/testdrivenP.html.

[21] S. Kim, J. A. Clark, and J. A. McDermid, “Investigating the Effectiveness of Object-Oriented Strategies with the Mutation Method,” in Mutation Testing for the New Century, Springer, Boston, MA, 2001, pp. 4–4.

[22] C. Wohlin, M. Höst, and K. Henningsson, “Empirical research methods in software engineering,” Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 2765, pp. 7–23, 2003, doi: 10.1007/978-3-540-45143-3_2.

[23] W. Zheng, “Automatic Software Testing Via Mining Software Data,” 2011.

[24] S. Rani, B. Suri, and S. K. Khatri, “Experimental comparison of automated mutation testing tools for Java,” Dec. 2015, doi: 10.1109/ICRITO.2015.7359265.

[25] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, “PIT: A practical mutation testing tool for Java (Demo),” ISSTA 2016 - Proc. 25th Int. Symp. Softw. Test. Anal., pp. 449–452, Jul. 2016, doi: 10.1145/2931037.2948707.

[26] H. Coles, “GitHub - pitest/pitclipse: Mutation testing for Java in Eclipse IDE. Based on PIT (Pitest).” https://github.com/pitest/pitclipse.

[27] H. Coles, “PIT - Mutation operators.” https://pitest.org/.