International Journal of Computer Science and Engineering (IJCSE)
ISSN(P): 2278-9960; ISSN(E): 2278-9979
Vol. 5, Issue 4, Jun - Jul 2016; 11-22

International Academy of Science, Engineering and Technology
Connecting Researchers; Nurturing Innovations

MUTATION TESTING TOOLS FOR JAVA PROGRAMS – A SURVEY

DANA H HALABI[1] & ADNAN SHAOUT[2]

[1]Department of Computer Science Princess Sumaya University for Technology Amman, Jordan

[2]Sabbatical from the ECE Department at the University of Michigan, Dearborn

ABSTRACT

Mutation testing is a white-box technique that can be used in software testing to ensure programs free from syntax errors. Systems based on Java are widely used because it is independent-platform language. When testing Java programs, you should keep in mind that Java is an object programming language which combine the features of procedural languages and the features of object oriented languages. In testing software all features should be accounted for. In this paper we will present a study for the Java program different Mutation Testing Tools and focus on the following characteristics: the mutation operators supported by a tool, mutant generation methods, speedup techniques, and other comparisons specifications. This survey will be useful for software engineers who are interested in testing and in developing new testing tools or expand existing ones.

KEYWORDS: Java, Mutation Operators, Mutatin Testing Tools, Object Oriented, Survey

INTRODUCTION

Java as an object oriented (OO) language that has many useful features, including: inheritance, encapsulation, polymorphism and others. These features enables developers to develop efficient systems in more flexible and systematic ways.

Java combines the features of procedural language and object oriented language. So the faults that may be detected in the Java programs will also include the faults related to procedural and OO languages [1]. For testing tools, the testing should apply to both the procedural and OO features [2, 3, 4 and 5].

Mutation testing [6] is a white-box fault-based testing technique that measures the effectiveness of test suite. Faults are generated against the original program by creating a set of mutant versions. These mutants are created from the original program by applying mutation operators, which generate syntactic changes to the programs. Then generate test cases and execute them against the original and mutant programs. The aim of testing is to produce an acceptable output that will discover the errors and faults in the original program [7, and 8].

To apply mutation testing to object-oriented programs, the developers of these tools have facilities existing for mutation operators, which were developed for procedural-language programs, to OO programs. They also developed class mutation operators to detect OO specific faults.

The rest of the paper is organized as follows: section 2: will present the mutation testing, section 3: will present the mutation testing in Java programs, section 4: will present the mutation testing tools For Java programs, and finally conclusion will be presented in section 5.

MUTATION TESTING

Mutation testing is a white-box fault injection technique used to finds errors and faults in a program source code [9]. The effectiveness of a test suite is measured by how much it can find errors. It is defined by the percentage of faults that can be detected by the test cases. The mutation testing create the mutants (a mutant is the original program with simple syntactic changes, see Table 1 for examples). These faults (mutants) are injected in the code through mutation operators, which are based on common programmers' mistakes in a programming language.

Table 1: Example of Mutant Code

Original Code	Mutant Code

if (**x > 5**) {
System.out.println("x > 5");
}
	if (**x < 5**) {
	System.out.println("x > 5");
	}

The purpose of mutation testing is to ensure that a test suite can detect and find all developers' mistakes when comparing the output of the original code with the output of the mutated code against the same test cases. According to the results, a mutant is considered as killed mutant when the output of mutant is differ from the output of original, but it is considered alive if they both have the same output. Then to handle the alive mutant, a new test case should be generate to find the fault or the mutant code is completely equivalent to the original code [10]. Table 2 shows an example for equivalent mutant.

Table 2: Example of Equivalent Mutant

Original Code	Equivalent Mutant Code

for (int i=0; i<10; i++)
{
System.out.println(i);
}
	for (int i=0; i!=10; i++)
	{
	System.out.println(i);
	}

The process of applying the mutation testing starts by constructing the mutants test program [11 and 12]. The detailed testing process is shown in Figure 1 [13].

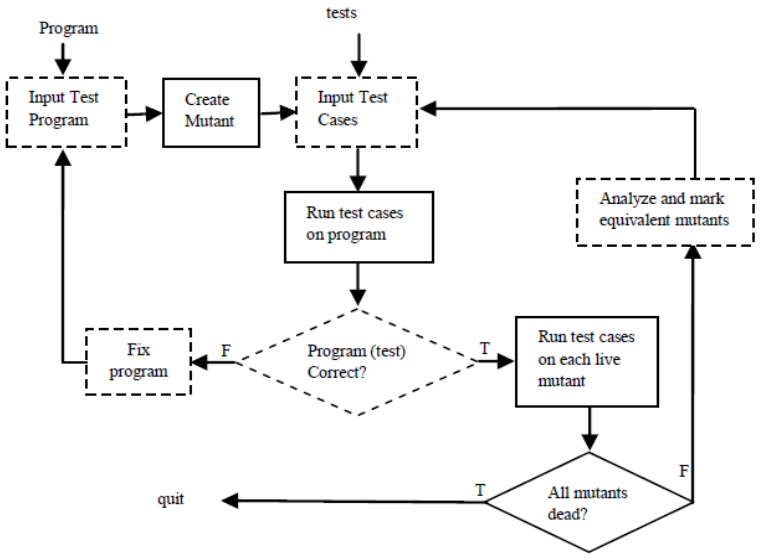

Figure 1: Mutation Testing Process [13]

MUTATION TESTING FOR JAVA PROGRAMS

To apply mutation testing to object-oriented programs (Java, C++, C#,..etc), mutation operators shouldn't be limited to traditional mutation operators of procedural language programs. It should support new mutation operators that apply mutation testing against the object oriented features that are class mutation operators to detect object oriented specific faults [14 and 15].

The two types of mutation operators are presented below:

Traditional Mutation Operators

The traditional mutation operators are presented in [16] and aims to supply the equivalence operators among different languages. The traditional mutation operators are based on the operators defined for Ada and Fortran, and they are used by Mothra tool [17]. These operators are shown in Table 3.

Table 3: Traditional Mutation Operators [18]

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Examples of some traditional mutation operators, are listed in Table 4.

Table 4: Examples of Some Traditional Mutation Operators

Operator	Description	Example
ABS	Absolute Value Insertion	a = b + c to a = 0
AOR	Arithmetic Operator Replacement	a = b + c to a = b − c
LCR	Logical Connector Replacement	a = b&c to a = b|c
ROR	Relational Operator Replacement	while(a < b) to while(a > b)
UOI	Unary Operator Insertion	a = b to a = −b

Class Mutation Operators

According to Java language features, there are four groups of class mutation operators [36]. These groups are as follows:

- Encapsulation
- Inheritance
- Polymorphism
- Java-Specific Features

The useful mutation operator is the operator that can handle all the possible syntactic changes for a programming language. Generally, the mutation operators can be created by one of the following ways [33 and 34]:

- delete,
- insert, and
- change a target syntactic element

For Java language, 29 class mutation operators were identified [36] for testing object-oriented and integration issues. The class mutation operators are listed in Table 5.

Table 5: Summary Class Mutation Operators for Java [36]

Language Feature	Operator	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	super keyword insertion
	ISD	super keyword

		deletion
	IPC	Explicit call to a parent's constructor deletion
Polymorphism	PNC	new method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCC	Cast type change
	PCD	Type cast operator deletion
	PRV	Reference assignment with other comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
Java-Specific Features	JTI	this keyword insertion
	JTD	this keyword deletion
	JSI	static modifier insertion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor deletion
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Examples of class mutation operators are listed in Table 6.

Table 6: Example of Some Class Mutation Operators

Category	Operator	Description	Example
Inheritance	AMC	Access Modifier Change	public Stack s; to private Stack s;
Polymorphism	PNC	new method call with child class type	a = new A(); to a = new B(); where B is subclass of A

Overloading	OAN	Argument number change	s.Push(0.5,2);
			to
			s.Push(2);

Java-specific	JTD	this keyword deletion	this.size = size;
			to
			size = size;

Common Programming Mistakes	EOA	Reference assignment and content assignment replacement	list2 = list1;
			to
			list2 = list1.clone();

According to the nature of object oriented languages, class mutation operator should be applied at different levels. The object oriented languages are class-based, so mutation operators should take care of mutation related to in-class and out-class language behavior [7 and 19]. These levels are presented as follows:

- Unit level: at this level apply traditional operators to function or method of class, checking its correctness.
- Class level: This level deals with the mutation of object oriented features.
- Integration level: Intermediate level between the unit and the system levels, checking the function invocations.
- Multi-class level: Operators at this level are intended to test a complete program: interactions among functions, classes, etc.

MUTATION TESTING TOOLS FOR JAVA PROGRAMS

An important characteristics of mutation testing tools are the mutation operators supported by a tool. Two types of mutation operators for object-oriented languages can be distinguished, as suggested by [20]. The types are presented as follows:

- Traditional mutation operators adapted from procedural languages and
- Object oriented (or class) mutation operators developed to handle specific features in object oriented programming.

Other characteristics of mutation testing tools are mutant generation methods (via bytecode or source code modification) and time reducing techniques (e.g., mutant schemata generation (MSG)) [21]. The MSG method is used to generate multi-one-mutant programs at the source level that each one can handle more than one mutants [22].

A comparison among Java mutation testing tools is listed in Tables 7.

Table 7: Comparison of Mutation Testing Tools for Java

Characteristic	Mujava	Mueclips	Jester	Jumble	RI	Judy	PIT
Open-source	Commercial	Open source	Open source	Opensource	Open source	Open source	Open source
OO mutation operators	Yes	Yes	No	No	No	Yes	No
Mutant generation level	Bytecode	Source code and byte code	Source code	Bytecode	NA	Source code	Byte code
Produces non-executable mutant	Yes	Yes	Yes	Yes	NA	Yes	Yes
Meta-mutant	Yes	Yes	No	No	Yes	Yes	NA
Mutants format	Separate class file	Separate source and class files	Separate source files	in-memory	NA	grouped in source files	Separate file in memory
JUnit support	No	Yes	Yes	Yes	Yes	Yes	Yes
Interface	GUI	Eclipse Plug-in	Command line	Eclipse Plug-in	NA	Command line	command line & Eclipseplugin
Support Batch Execution	No	No	Yes	No	NA	Yes	Yes
Speed	2	3	1	3	NA	1	1

Mujava

MuJava [22, 23 and 24] has a large set of traditional and class mutation operators specific for Java language. To speed up the mutant generation time, MuJava has implemented an advanced translation methods, 1) bytecode transformations and 2) MSG. Using these methods made it faster than using separate compilation approach [22]. Disadvantages of MuJava are as follows [25]:

- The final report of mutants are not shown in an appropriate way,
- The tool does not support batch mode, and
- It doesn't support JUnit test.

Muclipse

MuClipse [26] is a plugin within eclipse [27] for mutant generation and for JUnit testing. It is based on MuJava [23]. MuClipse supports 15 traditional operators and 28 class mutation operators. It has useful GUI which provide mutant viewer to view the mutant, and test case executor. After selecting the class to mutant, MuClips automatically generates mutants and for each mutant it runs JUnit test suites. Its report summarize the number of live and killed mutants with the mutation score for the test suite. MuClips generates the parse tree for a java program. The mutation operates on the source code, and should be compiled before running JUnit test suite. It doesn't run equivalent mutants.

Disadvantages of the tools MuClipse are as follows:

- It can't test the whole program by one-click,
- The test is done class by class and the user is unable to select more than one class at each time, and

- The test become more complex when the program has packages.

Jester

Jester [29] use oversimplified mechanism of mutant's generation. It is considered as a useful tool by some testers [28], but it is a very slow tool.

Actually, Jester depends on extending the default set of mutation operations, but this can cause problems related to performance and reliability. Jester supports limit range of mutation operators, based only on string substitution. The mutants are not strongly generated, and can lead to break the code.

Disadvantages [25] of Jester are as follows:

- It generate, compile and run unit tests against each mutant,
- It takes a long time to finish a run, and
- The final report can't be generated automatically.

Jumble

To reduce the testing time, Jumble [30] implemented bytecode transformation technique. It supports JUnit test. Jumble is used in testing industrial applications.

Jumble does not support all mutation operators. It is limited to the following operators: AOR, ASR, COR, LOR, ROR and SOR. The operators in Jumble are implemented such that only one of the mutations defined by the mutation operators is applied. For example, the AOR is implemented by replacing `{' with `+', and not by each of the other arithmetic operators (`*', `/', and `%').

Disadvantages [25] of Jumble are as follows:

- It does not support the class mutation operators, and
- It provides the fixed replacements of mutation operators, e.g. AOR mutant operator.

Response Injection (Ri)

RI is a prototype [31] based on the aspect oriented mechanisms to generate mutants. It has simple traditional mutation operators which are changing primitive types and two class mutation operators that apply to string class objects and null value to objects. RI checks only the result of a method, and throw exceptions.

Disadvantages of RI are as follows:

- It doesn't support mutants for testing objects passed by parameters of a method,
- The resources of documentation and source code are not available, and
- It could be better if it can use all the facilities of aspect oriented programming.

Judy

Judy [32] is implemented using Java and AspectJ extensions based on the FAMTA Light approach.

The most important features of Judy are as follows:

- The high performance of mutation testing process,
- Advanced mutant generation methods,
- Integration with professional development environment tools,
- Full automation of testing, and
- Support the latest version of Java. These features enable the tool to enforce mutation testing to most recent Java application.

Judy Supports the Following

- Traditional mutation operators: ABS, AOR, LCR, ROR and UOI. These operators are selected from procedural languages to minimize the number of mutation operators, and at the same time maximizing testing strength, and
- Class mutation operators of Java language, e.g. UOD, SOR, LOR, COR, and AS. Judy supports the EOA and EOC mutation operators [32] that model object-oriented faults that are difficult to detect [20].

PIT

PIT [35] is an open source mutation testing tool. It works fast for mutant generation. PIT has four phases: mutant generation, test selection, mutant insertion and mutant detection in PIT. PIT uses mutation operators like conditional boundary, negate conditionals, conditions removal, math mutator and more. Mutation is performed at byte code level. Initially, mutants are generated and test data are selected to run over mutants. Then mutants are loaded into the JVM and detected by the test set. Along with mutants details, it also generates line coverage and mutation coverage, thus requires some overhead.

CONCLUSIONS

It can be concluded that the MuJava, MuEclips and Judy are more efficient tools, as compared to the others since these tools support class mutation operators. Judy is by far the most efficient mutation testing tool because it supports batch execution that yields fast performance. It also perform the mutation at the bytecode level.

Regarding to GUI, every tool supports its own technique and there is a lack of common interface among the tools which makes it difficult to compare.

There are many factors other than the execution time that effects the performance of the presented tools. Such factors are as follows:

- Number of mutation operations each tool has (traditional and class mutations),
- The implementation for these operations (simple or complex, and
- Testing result reports.

There is a lake of information available about these factors from the presented tools that makes it difficult to determine their performance.

REFERENCES

1. J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In Proceedings of the 12th International Symposium on Software Reliability Engineering, pages 84–93, November 2001.
2. R. T. Alexander, J. M. Bieman, S. Chosh, and B. Ji. Mutation of Java objects. In 13th International Symposium on Software Reliability Engineering, pages 341–351, November 2002.
3. P. Chevalley and P. Th´evenod-Fosse. A mutation analysis tool for Java programs. Journal on Software Tools for Technology Transfer (STTT), pages 1–14, December 2002.
4. S. Kim, J. Clark, and J. McDermid. Class mutation: Mutation testing for object-oriented programs. OOSS: Object-Oriented Software Systems, October 2000.
5. Y. S. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for Java. Proceedings of 13th International Symposium on Software Reliability Engineering, November 2002.
6. R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. IEEE Computer, 11(4):34–41, November April. 1978
7. Marcio E. Delamaro, Marcos L. Chaim and Auri M. R. Vincenzi “Twenty five years of research in structural and Mutation testing”in 25th Brazilian Symposium on Software Engineering,978-0-7695-4603-2/11, IEEE 2011.
8. Fevzi Belli, MutluBeyazit“A Formal framework for mutation testing” in proceeding of: Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, 2010.
9. M. R. Woodward, “Mutation testing - its origin and evolution,” Information and Software Technology, vol. 35, no. 3, Mar. 1993, pp. 163–169. [Online]. Available: http://dx.doi.org/10.1016/0950-5849(93)90053-6
10. Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” Software Engineering, IEEE Transactions on, vol. 37, no. 5, Oct. 2011, pp. 649 –678.
11. Offutt A., A Practical System for Mutation Testing: Help for the Common Programmer, Twelfth International Conference on Testing Computer Software, 99-109, Washington D.C. June 1995.
12. DeMillo R., Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering, 17(9): 900-910, 1991.
13. Scholivé, M., Beroulle, V., Robach, C., Flottes, M.L., Rouzeyre, B., "Mutation Sampling Technique for the Generation of Structural Test Data", published in proceeding
14. Sunwoo Kim John A. Clark John A. McDermid“Class Mutation: Mutation Testing for Object-Oriented Programs” In the Proceedings of the FMES 2000. October 2000.
15. Y.-S. Ma, “Object-Oriented Mutation Testing for Java,” PhD Thesis, KAIST University in Korea, 2005
16. J. Boubeta-Puig, A. Garc´ıa-Dom´ınguez, and I. Medina-Bulo, “Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages,” in Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE. Berlin, Germany:

IEEE, 2011, p. 398–407, print ISBN: 978-1-4577-0019-4. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.52

17. K. N. King and A. J. Offutt, “A FORTRAN language system for mutation-based software testing,” Software - Practice and Experience, vol. 21, no. 7, 1991, pp. 685–718.
18. Yue Jia and Mark Harman, "An Analysis and Survey of the Development of Mutation Testing", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Digital Object Identifier 10.1109/TSE.2010.62, 0098-5589/10/$26.00 © 2010 IEEE
19. Y.-S. Ma, Y.-R. Kwon, and A. J. Offutt,“ Inter-class Mutation Operators for Java,” in Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE‟02). Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, p. 352.
20. Ma, Y.S., Harrold, M.J., and Kwon, Y.R.: `Evaluation of Mutation Testing for Object-Oriented Programs'. Proc. 28th Int. Conf. on Software Engineering, Shanghai, China, May 2006
21. Untch, R.H., Offutt A.J., and Harrold, M.J.: `Mutation Analysis Using Mutant Schemata'. Proc. Int. Symp. on Software Testing and Analysis, Cambridge, USA, June 1993, pp. 139-148
22. Ma, Y., Offutt, J., Kwon, Y.R.: `MuJava: an automated class mutation system', Software Test Verif Rel, 2005, 15, (2)
23. Offutt, J., Ma, Y.S., and Kwon, Y.R.: `An Experimental Mutation System for Java', SIGSOFT Software Eng. Notes, 2004, 29, (5)
24. Ma, Y.S., Offutt, J., and Kwon, Y.R.: `MuJava: A Mutation System for Java'. Proc. 28th Int. Conf. on Software Engineering, Shanghai, China, May 2006
25. Madhuri Sharma, Neha Bajpai, Automatic Generation and Execution of Mutants, International Journal of Computer Applications (0975 – 8887) Volume 44– No.3, April 2012
26. H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," in Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION,'07, 2007
27. Eclipse. [Online]. https://www.eclipse.org/
28. Moore, I.: `Jester - a JUnit test tester'. Proc. 2nd Int. Conf. on Extreme Programming and Flexible Processes in Software Engineering, Sardinia, Italy, May 2001
29. Offutt, J.: `An analysis of Jester based on published papers'. http://cs.gmu.edu/~o_utt/jester-anal.html, accessed September 2006
30. Irvine, S. A., Tin, P., Trigg L., Cleary, J. G., Inglis, S., and Utting, M.: `Jumble Java Byte Code to Measure the Effectiveness of Unit Tests'. Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques, Windsor, UK, September 2007
31. Bogacki, B., and Walter, B.: `Aspect-Oriented Response Injection: An Alternative to Classical Mutation Testing'. Proc. IFIP Work. Conf. on Software Engineering Techniques, Warsaw, Poland, October 2006

32. Judy, http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.Judy, accessed June 2007
33. Ma, Y.S., Kwon, Y.R., and Offutt, J.: `Inter-Class Mutation Operators for Java'. Proc. 13th Int. Symposium Software Reliability Engineering, Washington, USA, November 2002
34. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., and Zapf, C.: `An Experimental Determination of Sufficient Mutant Operators', ACM Trans. Software. Eng. and Meth., 1996, 5, (2)
35. Henry Coles. (2012) PIT Mutation Testing. [Online]. http://pitest.org/
36. Yu-Seung Ma and Jeff Offutt, " Description of Class Mutation Operators for Java", August 1, 2014