How do Java mutation tools differ?

DOMENICO AMALFITANO, Department of Electrical Engineering and Information Technology (DIETI), University of Naples Federico II
ANA C. R. PAIVA, Faculty of Engineering of the University of Porto & INESC TEC
ALEXIS INQUEL, ISEN Brest
LUÍS PINTO, Faculty of Engineering of the University of Porto
ANNA RITA FASOLINO, Department of Electrical Engineering and Information Technology (DIETI), University of Naples Federico II
RENÉ JUST, Computer Science & Engineering, University of Washington

Mutation analysis techniques have been successfully deployed over the last decade, including mutation-based testing in industrial settings. This success has been possible thanks to the effort spent by both academic and industrial communities in designing, developing, and comparing mutation tools. While some comparative studies exist in the literature, several tool aspects have not been taken into account. A more comprehensive comparison is desirable to enable users to make an informed choice. Such choice of mutation tool may depend on the specific use case. For instance, *Researchers* may find the configurability of a tool particularly relevant to support different studies, while *Educators* may prefer tools that are easier to install and use. Finally, *Practitioners* may require that a tool must integrate with the company's existing software development environment. This paper makes four main contributions: (1) it reports on a meta-analysis of existing comparisons of Java mutation tools; (2) it proposes a comprehensive mutation tool comparison framework encompassing five dimensions (*Version*, *Deployment*, *Mutation process*, *User-centric features*, and *Mutation operators*), each with multiple attributes; (3) it reports on an application of the proposed framework to eight Java mutation tools, involving a literature survey, a student survey, and a tool-authors survey; (4) it reports on a survey to researchers, educators, and practitioners that identifies their key considerations for selecting a mutation tool. Taken together, this paper proposes a framework that can aid future tool comparisons and shows how the framework can guide the selection of a suitable mutation tool for a given use case.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: mutation testing, mutation testing tools analysis, mutation testing tools comparison

ACM Reference Format:
Domenico Amalfitano, Ana C. R. Paiva, Alexis Inquel, Luís Pinto, Anna Rita Fasolino, and René Just. 2022. How do Java mutation tools differ?. *Communications of the ACM* 1, 1 (January 2022), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Domenico Amalfitano, domenico.amalfitano@unina.it, Department of Electrical Engineering and Information Technology (DIETI), University of Naples Federico II, Via Claudio, 21, Naples, Italy, 80125; Ana C. R. Paiva, apaiva@fe.up.pt, Faculty of Engineering of the University of Porto & INESC TEC, Rua Dr. Roberto Frias, s/n, Porto, Portugal, 4200-465; Alexis Inquel, alexis.inquel@isen-ouest.yncrea.fr, ISEN Brest, 20 Rue Cuirassé Bretagne, Brest, France, 29200; Luís Pinto, up201809188@fe.up.pt, Faculty of Engineering of the University of Porto, Rua Dr. Roberto Frias, s/n, Porto, Portugal, 4200-465; Anna Rita Fasolino, fasolino@unina.it, Department of Electrical Engineering and Information Technology (DIETI), University of Naples Federico II, Via Claudio, 21, Naples, Italy, 80125; René Just, rjust@cs.washington.edu, Computer Science & Engineering, University of Washington, Seattle, WA, USA.

2022. /2022/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Back in 2011, Jeff Offutt noted that *"The field of mutation analysis has been growing, both in the number of published papers and the number of active researchers."* [33]. This trend has since continued, as confirmed by a recent literature survey [36].

Mutation analysis is "the use of well defined rules defined on syntactic descriptions to make systematic changes to the syntax or to objects developed from the syntax" [33].

Mutation analysis has been successfully used in research for assessing test efficacy and as a building block for testing and debugging approaches. It systematically generates syntactic variations, called *mutants*, of an original program based on a set of *mutation operators*, which are well defined program transformation rules. The most common use case of mutation analysis is to assess test efficacy. In this use case, mutants represent faulty versions of the original program, and the ratio of detected mutants quantifies a test suite's efficacy. Empirical evidence supports the use of systematically generated mutants as a proxy for real faults [2, 5, 19]. Another use case is automated debugging (e.g., [10, 11]). In this use case, mutants represent variations of a faulty program and are used either to locate the fault or to iteratively mutate the program until it satisfies a given specification (e.g., passes a given test suite).

Mutation analysis can be applied at different levels, including design and specification level, unit level, and integration level. Similarly, it can be applied to both models and programs. For example, prior work applied mutation analysis at the design level to Finite State Machines, State Charts, Estelle Specifications, Petri Nets, Network protocols, Security Policies, and Web Services [16].

Mutation-based testing leverages mutation analysis and is a testing approach that uses mutants as test goals to create or improve a test suite. Mutation-based testing has long been considered impractical because of the sheer number of mutants that can be generated, even for small programs. Mutation-based testing is now increasingly adopted in industry, in part due to a shift in perspective, including the notion of incremental, commit-level mutation, suppression of unproductive mutants, and the focus on individual mutants as opposed to adequacy w.r.t. mutant detection [3, 37–39].

The software engineering community has devoted great effort to developing mutation tools, which are available for a variety of programming languages, including Java, Python, Javascript, C#, Ruby, and PHP [16, 36]. However, existing tools, even for the same programming language, differ substantially. For example, different tools implement different mutation operators, applied to source code or byte code, and produce different output artifacts, such as mutated source code and a mutant-test kill matrix. These differences and their corresponding trade-offs for different use cases are not always apparent, often due to a lack of documentation and/or empirical evidence. As a consequence, a researcher, educator, or practitioner may make suboptimal choices or may be forced to conduct a deeper investigation when *choosing the most suitable tool for the use case at hand.*

This paper characterizes the empirical studies that analyzed and compared Java mutation tools, based on a rapid review of the research literature. Additionally, this paper proposes a framework for comparing mutation tools, considering five dimensions: Tool version; Deployment; Mutation process; User-centric features; and Mutation operators. Finally, this paper uses the proposed framework to highlight the similarities and differences of 8 state-of-the-art Java mutation tools.

The rest of the paper is organized as follows. Section 2 describes background material. Section 3 summarizes prior studies that compared Java mutation tools, using a rapid review process. Section 4 describes our proposed comparison framework, and Section 5 details an application of this framework for 8 Java mutation tools. Section 6 details the considerations associated with choosing a suitable mutation tool for use cases in research, education, and practice. Finally, Section 7 concludes and outlines directions for future work.

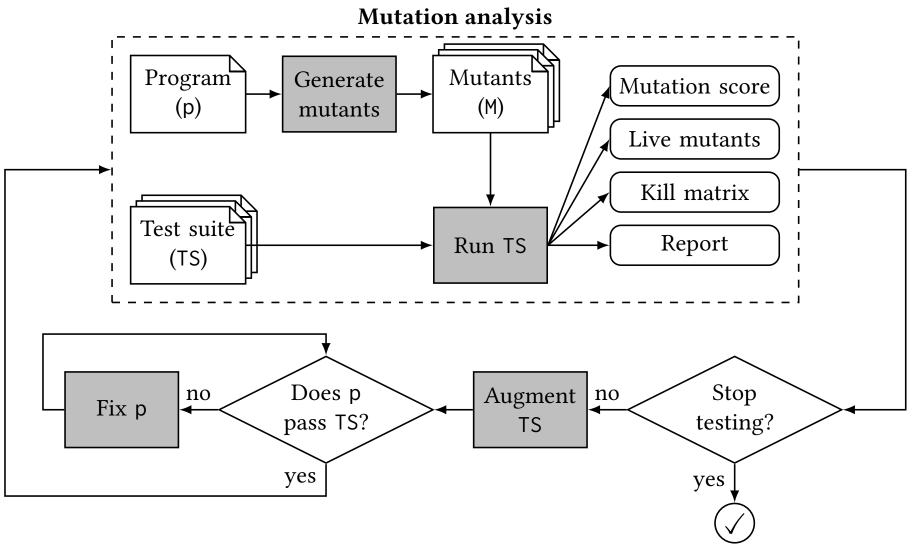


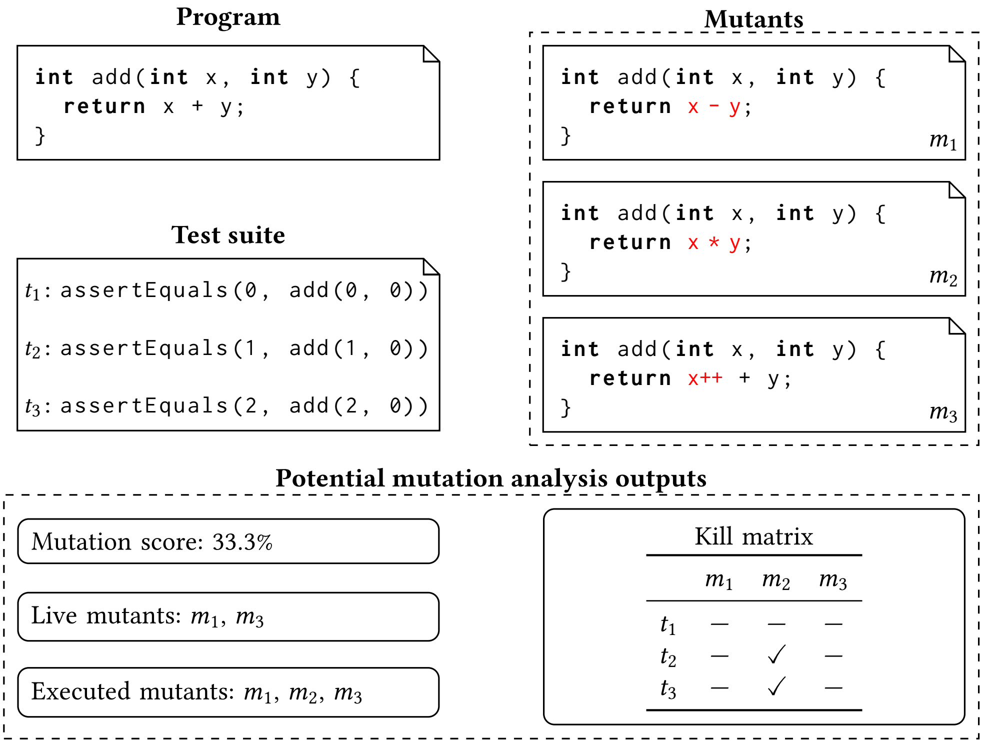
Fig. 1. Mutation-based testing.

2 BACKGROUND

Figure 1 visualizes a common mutation analysis process and how mutation-based testing is a specific use case and instantiation of that process. An interested reader can find related process descriptions in [36] and [16], which are, like ours, an adaptation of Offutt's and Untch's original formulation of mutation analysis [32]. The literature has largely used the terms mutation analysis and mutation testing interchangeably, but we make the distinction more precise because other mutation-based approaches and use cases exist (e.g., test suite reduction, fault localization, or program repair). To avoid ambiguity, we use the terms mutation analysis and mutation-based testing. Mutation analysis involves two main steps—mutant generation and test suite execution. Mutation-based testing iteratively applies mutation analysis, until a stopping condition is met, and involves two additional steps—test suite augmentation and (possibly) program repair.

As an example, consider Figure 2: a mutation analysis, with an original program and a corresponding test suite. First, the analysis generates the three mutants (m_1–m_3), each by applying a mutation operator to the return statement of the original program. Next, the analysis executes each test against each mutant and computes the kill matrix shown in the lower-right corner. A test that detects a mutant is said to *kill* that mutant. A mutant that is not killed by any test is referred to as a *live* mutant. Finally, the analysis reports on the results, indicating the mutation score, the set of live mutants, and the kill matrix. While the mutation score is usually defined as the ratio of killed to all non-equivalent mutants, most tools approximate it and report the number of killed mutants divided by the total number of generated mutants. The reason for this is that the set of equivalent mutants is unknown and reasoning about program equivalence is an undecidable problem. Note that the computation of a complete kill matrix is not required for all use cases. For example, if the goal is to simply compute the mutation score and the set of live mutants, then t_3 need not be executed against m_2 after t_2—m_2 is already known to be killed at that point. Indeed, a kill matrix is rarely, if ever, computed in mutation-based testing because it is computationally expensive.

The mutation analysis results in Figure 2 show that two out of three mutants are live and that t_1 does not kill any mutants. Generally speaking, a test suite that fails to kill most of the mutants is



```
int add(int x, int y) {
  return x + y;
}
```

```
t1: assertEquals(0, add(0, 0))
t2: assertEquals(1, add(1, 0))
t3: assertEquals(2, add(2, 0))
```

```
int add(int x, int y) {
  return x - y;
}
```

```
int add(int x, int y) {
  return x * y;
}
```

```
int add(int x, int y) {
  return x++ + y;
}
```

	m_1	m_2	m_3
t_1	–	–	–
t_2	–	✓	–
t_3	–	✓	–

Fig. 2. Mutation analysis example.

deficient and should be improved. The core idea of mutation-based testing is to use live mutants as concrete test goals. In the example in Figure 2, mutant m_1 is a live mutant and indicates that the test suite lacks a test case—one that passes a non-zero argument to the second parameter of the add method. Mutation-based testing repeats this iterative process of adding tests based on mutants until a stopping condition is met, e.g., a given mutation score threshold or a fixed test budget.

Not every mutant can be killed. An *equivalent mutant* is semantically equivalent to the original program and cannot be killed by any test. Mutant m_3 in Figure 2 is an example of an equivalent mutant. Moreover, not every mutant that can be killed should be killed. Traditionally, killable mutants were generally deemed desirable because they lead to tests; conversely, equivalent mutants were generally deemed undesirable. Petrović et al. [39], however, noted that this classification is unworkable in practice and insufficient to capture the notion of developer productivity. For example, developers justifiably should not and, in practice, will not write a test for a killable mutant if that test would be detrimental to the test suite quality, in particular maintainability. Conversely, equivalent mutants may point to actual program issues, prompting developers to make meaningful improvements to the code itself. Petrović et al. introduced the notion of productive mutants. A mutant is productive if the mutant is killable and elicits an effective test, or if the mutant is equivalent but its analysis and resolution improves code quality. For example, a mutant that changes the initial capacity of a Java collection (e.g., replacing `new ArrayList(20)` with `new ArrayList(10)`) is unproductive. While such a mutant is theoretically killable by writing a test that asserts on the collection capacity or expected memory allocations, it is unproductive to do so because the corresponding test would be brittle and not testing actual functionality. Note that the notion of productive mutants is qualitative: different developers may sometimes reach different conclusions as to whether a test is effective.

3 RAPID REVIEW OF JAVA MUTATION TOOL COMPARISONS

Based on the knowledge of what previous comparative studies propose, we decided to collect evidence from the literature to understand (1) how mutation tools were compared and (2) whether some mutation tools consistently outperform others along multiple dimensions.

We adopted a Rapid Review (RR) [41] process for this purpose. RRs are literature review processes less formal than Systematic Mappings (SMs) and Systematic Literature Reviews (SLRs), but similarly they follow a well-structured selection process. Hence, RRs can be further analyzed, replicated, and improved by other researchers. According to Cartaxo et al. [4], the main goal of a RR is to reduce the amount of time needed to gather, analyze, interpret, review, and publish evidence that could benefit practitioners. To achieve this goal, RRs deliberately omit or simplify steps of traditional SLRs (e.g. limiting literature search, using just one person to screen studies, skipping formal synthesis). Our RR process relies on the following 4 sequential steps:

(1) Scopus search string definition and application.
(2) Primary studies selection.
(3) Data extraction.
(4) Data analysis and abstraction.

3.1 Scopus search string definition and application

We defined the search string, specifically crafted for the Scopus[1] search engine, as follows:

```
TITLE-ABS-KEY ((“Mutation testing” OR “Mutation analysis” OR “mutation
tool*” OR “mutation testing tool*”) AND (“Empirical study” OR
“Empirical evaluation” OR “empirical analysis” OR “compar* mutation*”
OR “Experimental comparison”)) AND (LIMIT-TO( SUBJAREA, “COMP”))
```

The key rationale is to search for studies in the area of computer science that present an empirical evaluation or an experimental comparison of mutation tools. We applied this search string to the Scopus engine in November 2020, and it returned 187 results.

3.2 Primary studies selection

Our goal was to select primary studies presenting an empirical evaluation for comparing two or more Java mutation tools. To that end, we divided the 187 search results into 2 sets of 93 and 94 studies. Two of the authors independently analyzed the studies of the two sets. Specifically, each researcher identified papers that satisfied all of the following four inclusion criteria:

I_1 : The study presents an empirical study involving at least two Java mutation tools.
I_2 : The study considers Java mutation tools that are publicly accessible and free of charge.
I_3 : The study analyzes Java mutation tools that are described in at least one publication.
I_4 : The authors of the study are different from the authors of the analyzed mutation tools.

The selection process resulted in 5 primary studies, which are subsequently referred to as S1 [22], S2 [12], S3 [28], S4 [40], and S5 [8].

[1]https://www.scopus.com/search/form.uri

Table 1. Analyzed mutation tools and study design for each of the five primary studies.

	Primary Study				
	S1 [22]	**S2 [12]**	**S3 [28]**	**S4 [40]**	**S5 [8]**
Compared Mutation Tool					
PIT [7]	✓	✓	✓	✓	✓
MuJava* [25]	✓		✓	✓	✓
Major [21], [17]	✓	✓	✓		✓
Jumble [15]		✓		✓	✓
Javalanche* [42]		✓			✓
Jester* [30]				✓	✓
Judy [26]		✓		✓	✓
Bacterio* [29]					✓
Test Case Generation					
Manual	✓			✓	
Automated	✓				
Existing test cases		✓			✓
Adopted Metric					
Real fault detection	✓				
Mutual killability	✓		✓		
Disjoint mutation score	✓				
Correlation measures		✓			
Mutation score		✓	✓	✓	✓
Number of generated mutants		✓	✓	✓	✓
Number of equivalent mutants	✓	✓	✓		
Number of killed mutants	✓				✓
Number of generated test cases	✓		✓		
Execution time				✓	✓
Subjects					
Real-world project	5	25			6
Simple Java class	12		4	4	

*The tool presented limitations when used to mutate real-world projects.

3.3 Data extraction

One of the authors fully read the five primary studies, and extracted sentences to collect evidences for (1) which Java mutation tools were analyzed and (2) how these tools were compared. These sentences were stored in a spreadsheet file for analysis.

3.4 Data analysis and abstraction

We adopted a Delphi method, which is commonly used when the problem under analysis can benefit from collective and subjective judgments or decisions and when group dynamics do not allow for effective communication (e.g., time differences, distance) [14]. Three of the authors, in weekly meetings, iteratively analyzed the extracted data, resolved ambiguity, and converged onto the final abstraction shown in Table 1. Based on a final data analysis, we made three key observations.

Observation 1: the five primary studies compared a total of 8 Java mutation tools. Table 1 lists these tools together with the references that report on how these tools are implemented and how they are used. As shown in Table 1, PIT is the only tool analyzed in all five studies, followed by Major and MuJava (four studies), Jumble and Judy (three studies), Jester (two studies), and Bacterio (one study).

Ensuring that our analysis does not miss other Java mutation tools, we executed an additional query in the Scopus database, using the following search string:

```
TITLE-ABS(("mutation tool*" OR "mutation testing tool*") AND java) AND
(LIMIT-TO(SUBJAREA, "COMP"))
```

This query searches more broadly for Java mutation tools presented in literature. We found three additional Java-specific mutation tools: HOMAJ [34], *Paraμ* [27], and JavaMut [6]. We did not include these three tools for two main reasons. First, no empirical comparison considered these tools. Second, these tools are not available. To the best of our knowledge, we can conclude that our list of mutation tools, reported in Table 1, is representative of the state of the art w.r.t. available Java mutation tools.

Observation 2: the empirical studies that compared the mutation tools used different study designs and measures. More concretely, the studies differ in three main aspects for evaluating the tools: (1) how the test cases for killing mutants were generated, (2) what evaluation metrics were adopted, and (3) what Java subjects were selected. We observed three distinct approaches for test case generation: (1) manually writing test cases, (2) automatically generating test cases, using tools such as EvoSuite [9] or Randoop [35], and (3) using existing test cases (i.e., using the test cases that are distributed with the subject application). Further, we observed a total of 10 adopted metrics, 9 of which evaluate the effectiveness and 1 evaluates the efficiency of the mutation tools. Almost all of the studies reported on absolute measures such as the mutation score, number of mutants, and number of test cases. *S1*, *S2*, and *S3* reported on additional measures:

(1) *Real fault detection*: measures fault-coupling [19, 31] and whether mutation-adequate test cases also detect real faults.
(2) *Mutual killability*: measures to what extent mutation-adequate tests derived from one tool kill all the non-equivalent mutants of another tool.
(3) *Disjoint mutation score*: measures how a tool performs compared to a reference mutant set.
(4) *Correlation measures*: correlates the absolute measures obtained when evaluating each tool independently.

Only two studies considered the efficiency of the mutation tools. In both cases, the costs of adopting the mutation tools were evaluated in terms of mutation analysis execution time. The five primary studies either considered simple Java classes or real-world projects from the Defects4J benchmark [18].

Observation 3: only 4 of the 8 tools (PIT, Major, Jumble, and Judy), did not present any limitations when executed on real-world projects. For example, prior studies excluded mutation tools from parts of their empirical evaluations because of tool limitations (see *S1*, *S2*, and *S5* for further details).

Given the diversity of study designs and measures, there is no clear evidence that one of the eight tools consistently outperforms the others—in particular when considering different use cases (see Section 6). While PIT and Major overall achieve slightly better results in most of the empirical evaluations, there is insufficient information to compare the tools for different use cases.

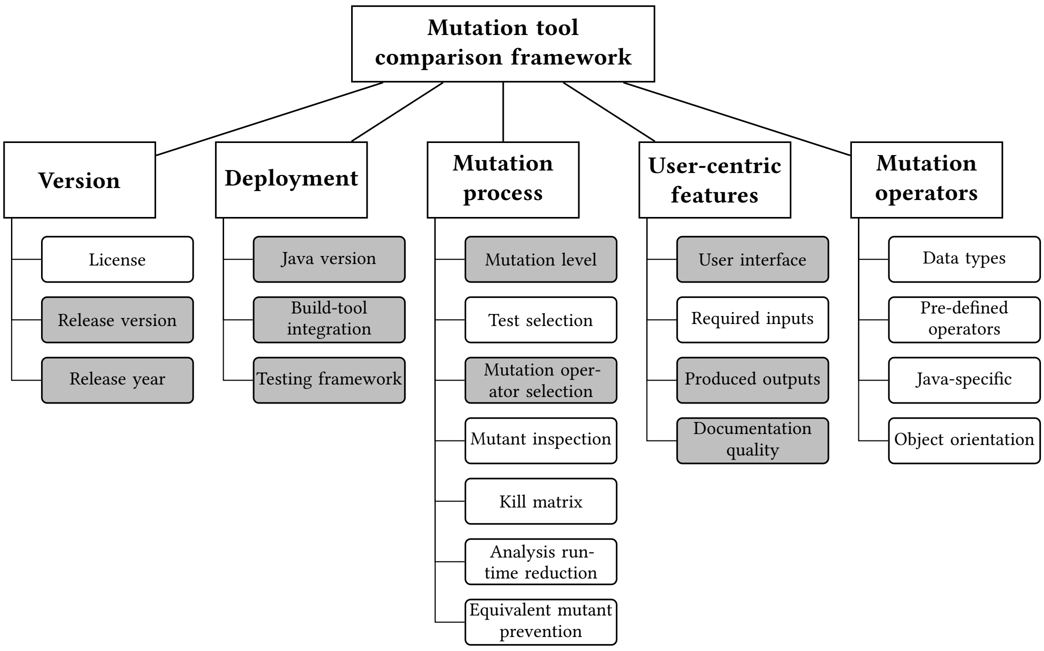


Fig. 3. Framework for comparing mutation tool features. Gray boxes indicate features reported in prior works.

4 MUTATION TOOL COMPARISON FRAMEWORK

The analysis of the selected papers showed that Java mutation tools were compared from the point of view of the features they offer. To provide a comprehensive, unified representation of the different ways the Java mutation tools can be qualitative compared according to their features, we inferred the *Mutation tool comparison framework* shown in Figure 3. This model describes each tool along five dimensions, each one with one or more attributes. The gray boxes represent dimensions or attributes that were already used as comparative parameters in the primary studies we analyzed, meanwhile the white boxes render the novel dimensions and attributes we introduced to provide additional details for comparing mutation tools. Overall, we introduced 11 novel attributes for a total of 21 attributes across 5 dimensions.

4.1 Version

This dimension characterizes the version of the tool and provides some indication about its level of obsolescence. It has the following 3 attributes:

(1) *License:* Type of license defining the terms and conditions for using, reproducing, and distributing the tool. It is a novel attribute and was introduced since we believe that this information can be useful for both practitioners and researchers who want apply the tool in their work.
(2) *Release version:* Number of the latest stable version of the tool.
(3) *Release year:* Date of the latest stable release of the tool.

4.2 Deployment

This dimension typifies the requirements of the execution environment where the mutation tool can be installed and executed. It has the following 3 attributes:

(1) *Java version:* the Java version that is compatible with the correct execution of the tool.

(2) *Build-tool integration:* the build tool(s) (i.e., maven and/or ant) needed for deploying and running the tool.
(3) *Testing framework:* the testing frameworks that should be installed on the running environment.

4.3 Mutation process

This dimension describes the features provided by the tool in supporting the execution of mutation-analysis processes. It has the following 7 attributes:

(1) *Mutation level:* it represents where the mutation operators are applied, i.e., source code or byte code.
(2) *Test selection:* how the tool allows the selection of the tests to be executed.
(3) *Mutation operator selection:* how the tool enables to select the mutation operators to apply.
(4) *Mutant inspection:* how the tool aids the inspection of the executed mutation operators.
(5) *Kill matrix:* it describes the characteristics of the kill matrix, i.e., which tests kill each mutant.
(6) *Equivalent mutant prevention:* strategies used to help identifying equivalent mutants.
(7) *Analysis runtime reduction:* strategies applied to reduce the cost of mutation analysis.

4.4 User-centric features

This dimension describes the "pick and use" characteristics of the tool. It has the following 4 attributes:

(1) *User interface:* the user interface for interacting with the tool.
(2) *Required inputs:* the artifacts needed as input for launching the tool.
(3) *Produced outputs:* the artifacts produced by the tool.
(4) *Documentation quality:* the quality of the online documentation for guiding the tool execution.

4.5 Mutation operators

This dimension expresses the tool's ability to implement different classes of mutation operators. Due to the lack, in all the primary studies we analyzed, of a unified approach for describing the operators actually implemented by each tool, we decided to abstract a set of reference mutation operator classes according to the official Java documentation[2]. This dimension has the following 4 attributes to represent classes of mutation operators:

(1) *Data types:* mutation operators for variables, constants, and literals.
(2) *Pre-defined operators:* mutation operators for pre-defined operations in Java.
(3) *Java-specific:* mutation operators for Java-specific language features.
(4) *Object orientation:* mutation operators for object-oriented language features.

5 FRAMEWORK APPLICATION TO JAVA MUTATION TOOLS

Our goal was to describe the 8 mutation tools according to the proposed framework. To that end, we first abstracted all the possible values that can be assumed by the framework attributes and then we outlined the mutation tools according to these values. We used a process that involved three different surveys for inferring these values: *literature and documentation survey*, *student survey*, and *tool-author survey*. Table 2 shows, for each framework attribute, which survey(s) we adopted for inferring its possible values. We used the literature and documentation survey to assess all attribute values for each tool. Additionally, we used the student and tool-author surveys to (1) validate and improve the attribute values inferred by the literature and documentation survey and (2) infer missing information.

[2]https://docs.oracle.com/en/java/

Table 2. Surveys adopted for recasting the framework

	Adopted survey		
	Literature and documentation survey	Student survey	Tool-author survey
Version			
License	✓	✓	
Release version	✓	✓	
Release year	✓	✓	
Deployment			
Java version	✓	✓	
Build-tool integration	✓	✓	
Testing framework	✓	✓	
Mutation process			
Mutation level	✓		✓
Test selection	✓	✓	
Mutation operator selection	✓	✓	✓
Mutant inspection	✓	✓	
Kill matrix	✓	✓	
Equivalent mutant prevention	✓		✓
Analysis runtime reduction	✓		✓
User-centric features			
User interface	✓	✓	
Required inputs	✓	✓	✓
Produced outputs	✓	✓	✓
Documentation quality	✓	✓	
Mutation operators			
ALL	✓		✓

5.1 Literature and documentation survey process

This process was performed in two steps. First, we performed a snowballing procedure [43]. Starting from the primary studies reported in Table 1, we gathered from the literature additional published papers describing in detail the selected tools (how they work, how they can be used for mutation analysis, and how they have been designed and implemented). Additionally, we consulted the tools' official documentation (user manual, technical report, etc.). Afterwards, we followed a Delphi-type cycle during which three researchers read the collected documents, classified the tools based on the framework, and explained their judgment.

5.2 Student survey process

We performed a user study with 46 MSc students in computer science. The user study involved an exit survey and had three main steps: (1) The first step involved a full, theoretical lecture (90 minutes) on mutation analysis and mutation-based testing. After this lecture, the students were divided in groups of 2 or 3 members and assigned a mutation tool. (2) The second step was designed to provide a hands-on training session for employing a mutation tool, using a simple Java class. During this step, an instructor provided guidance and focused on consolidating the students' theoretical knowledge, resolving open questions, and addressing any problems reported by the students. The simple Java class served as a didactic example to bring the students up to speed on using a mutation tool. (3) The third step aimed at assessing the tools' characteristics and infer some attributes by running the tools. After providing sufficient background and preparation, the goal of this step was to assess to what extent the students were able to install and use a mutation tool for a more complex Java class, relying on the provided documentation. To avoid bias, we randomly assigned a new tool to each group of students.

No student had experience with mutation analysis and mutation tools, prior to the lecture in the first step. The Java class used in the second step was `Triangle.java`, which contains a single method with three integer parameters, representing the lengths of a triangle's sides. This method's return value indicates the type of the triangle. At the end of the second step, all students successfully used the assigned tool and applied the learned concepts related to mutation-based testing. All tools described in Table 2, except Jester, were used in this step. The main reason is that Jester requires mutation operators to be provided in a configuration file, which was out of the scope of this work. Instead of asking students to use Jester, one of the authors installed the tool and used it to assess its attributes. Consequently, Jester was excluded from the third step. The third step randomly assigned a new mutation tool to each group and asked them to use these tools to perform mutation-based testing over three classes from the Defects4J benchmark: Cli (`org.apache.commons.cli.HelpFormatter`), Gson (`com.google.gson.stream.JsonWriter`) and Lang (`org.apache.commons.lang.time.DateUtils`). We aimed at using PIT, MuJava, Major, Jumble, Judy, and Bacterio for the third step, but had to exclude MuJava and Bacterio because we were unable to run them on the Defects4J classes. MuJava gave errors when trying to generate mutants. Bacterio was unable to execute the test cases.

The students installed the tools in a predefined environment/configuration: Oracle Virtual Machine on Ubuntu 20.04, using Java version 1.8 and JUnit 4. After installation, students had to create mutants for the class defined, perform mutation analysis to determine the number of killed and live mutants, and perform mutation-based testing to develop additional test cases to kill live mutants. At the end of the user study, students had to deliver a report and complete an exit survey[3] with 9 questions. Each question evaluated a tool characteristic on a scale of 1 to 5 and elicited a justification for the choice made. The final question allowed providing additional information.

5.3 Tool-author survey process

We implemented a survey[4] in Google Forms and send it to the authors to describe their tool according to the proposed framework. We designed this survey to collect information that was neither clearly reported in the tool's online documentation nor directly inferable by using the tool. As such, the authors' answers complemented the data extracted from the other surveys. By analyzing the answers we were able to (1) validate the data extracted in literature survey and (2) collect additional information we were not able to find elsewhere. In case, discrepancies arose, the author survey overrode our initial, partial findings. The survey included 7 specific and 4 open-ended questions. The 7 specific questions had an *Other* field where the respondents were free to extend the options we provided as possible answers. Three of the 4 open-ended questions were proposed to have more information about the Equivalent mutant prevention, Analysis runtime reduction, Required inputs, and Produced outputs. The Mutation operators implemented by the tools were

[3]https://forms.gle/V2knNg2eM3CeQvxX9
[4]https://forms.gle/GX3Ay3dpE6Tdbwop9

inferred by means of the literature and author surveys. The last open-ended question elicited suggestions and comments about our research.

5.4 Tools description according to the inferred attribute values

The data collected through the surveys were merged in a single spreadsheet file and analyzed by three researchers in weekly meetings where they analyzed and discussed the gathered information. The attribute values were inferred after unanimous consent was reached. Table 3 shows the values of the attributes we abstracted and the descriptions of the 8 mutation tools according to them.

5.4.1 Version. As for the *License* attribute we observed that 6 tools have a type permissive free software license, i.e. Apache2, GPL, or LGPL. Jester and Bacterio have been considered as freeware since they provide only an executable `.jar` file that can be freely used but they do not distribute the source code. Regarding the Release version, all the tools rely on a version control system, only Javalance does not provide releases tracking. As for the Release version most of the tools have not changed for five or more years (MuJava, Jumble, Javalanche, Jester and Judy). PIT, Major and Bacterio have very recent updates.

5.4.2 Deployment. Regarding the *Java version*, we observed that PIT, Major, Jumble and Judy were able to work with no limitations on large scale projects developed in Java 1.8, MuJava and Bacterio run only on simple Java classes developed in Java 1.8. When executed on real projects, MuJava presented unhandled exception, whereas the behavior of Bacterio was unreliable, i.e., it produced different mutants in diverse executions performed on the same code. We were not able to run Javalanche with Java 1.8 and so, we used Java 1.6. However, even with Java 1.6, students experienced lots of problems with the installation process and could not generate mutants neither run test cases. Jester was not used in our experiments so, we do not have practical evidence that it works on large scale projects. However, it worked with Java 1.8 in small project.

As for the *Build-tool integration* attribute, we observed that Javalanche and Jester need additional tools on the running environment. Javalanche requires Ant to compile, assemble, test and run both the mutation tool and the test cases. Jester relies on Python for running its scripts. All the remaining tools are self-contained, and some of them, like PIT, Major, and Jumble, exploit optional additional tools for extra features. PIT supports also several build tools like Maven, Ant, and Gradle. It can be also installed as Eclipse plugin, like Jumble, and IntelliJ plugin. Major is distributed along with its own Ant version. As for the *Testing framework*, all the tools need JUnit for running the test cases. PIT is the only one supporting also test cases developed in TestNG. Almost all tools support the automatic execution of test cases developed in JUnit 4, except for Jester and Bacterio that work only with JUnit 3.

5.4.3 Mutation process. Regarding the *Mutation level*, 5 out of 8 tools work exclusively at byte code level, whereas 2 tools, i.e., MuJava and Major, apply mutants on both source code and byte code level. Only Jester works at source code level.

As for the *Test selection*, PIT, Major and Javalanche provide an automatic mechanism that selects test cases to execute based on code coverage. For the other 5 tools, the tester selects manually the tests to be performed. In Jester and Judy the tester must remove the tests that should not be run from the folder where they are placed. In Jumble the JUnit tests to run should be listed trough the command line. MuJava and Bacterio allow to select the tests through the GUI.

Concerning the *Mutation operator selection*, we observed that 6 over 8 tools can be configured for applying selected mutation operators even if they belong to different classes. Bacterio and Jumble are less configurable since they are able to execute all the operators belonging to selected classes, MuJava provides both options.

Table 3. Tools described according to Versioning, Deploying, Mutation Process, and User-Centric Features

	Mutation tool							
	PIT	**MuJava**	**Major**	**Jumble**	**Javalanche**	**Jester** *	**Judy**	**Bacterio**
Version								
License	Apache 2	Apache 2	Apache 2/GPL	GPL	LGPL	Freeware	BSD	Freeware
Release version	1.5.0	4.0	1.3.5	1.3.0	None	1.3.7	3.0.0	3.0
Release year	2020	2015	2019	2015	2012	2005	2017	2019
Deployment								
Java version	Java 1.8	Java 1.8 **	Java 1.8	Java 1.8	Java 1.6 **	Java 1.8 **	Java 1.8	Java 1.8 **
Build-tool integration	Optional	Self-contained	Optional	Optional	Mandatory	Mandatory	Self-contained	Self-contained
Testing framework	JUnit 4 & TestNG	JUnit 4	JUnit 4	JUnit 4	JUnit 4	JUnit 3	JUnit 4	JUnit 3
Mutation process								
Mutation level								
Byte code	✓	✓	✓	✓	✓	–	✓	✓
Source code	–	✓	✓	–	–	✓	–	–
Test selection								
Automated	✓	–	✓	–	✓	–	–	–
Manual	–	✓	–	✓	–	✓	✓	–
Manual (GUI)	–	✓	–	–	–	–	–	✓
Mutation operator selection								
Operators	✓	✓	✓	–	✓	✓	✓	–
Operator classes	–	✓	✓	✓	–	–	–	✓
Mutant inspection								
Over LOC	✓	–	✓	–	✓	✓	✓	–
Over Live Mutants LOC	–	–	–	✓	–	–	–	–
Side-by-side	–	✓	–	–	–	–	–	✓
Kill matrix								
Test method	–	✓	✓	–	–	–	–	✓
Test class	–	✓	✓	✓	–	–	–	✓
Analysis runtime reduction								
Code coverage	✓	✓	✓	–	–	N/A	✓	✓
Tests order	✓	✓	✓	–	–	N/A	✓	✓
Parallel execution	–	–	✓	–	✓	N/A	✓	–
Mutant ranking	–	–	✓	–	–	N/A	–	✓
Infinite loop prediction	✓	–	✓	–	–	N/A	✓	–
Limited mutants	–	–	–	✓	–	N/A	–	–
Equivalent mutant prevention	Statistical	–	Code-based	Statistical	Code-based	–	–	–
User-centric features								
User interface								
CLI	✓	✓	✓	✓	✓	✓	✓	–
GUI	–	✓	–	–	–	–	–	✓
IDE	✓	✓	✓	✓	–	–	–	–
Required inputs								
Byte code	✓	–	–	✓	✓	–	✓	✓
Source code	✓	✓	✓	–	–	✓	–	–
Test cases	✓	✓	✓	✓	✓	✓	✓	✓
Produced outputs								
Mutated source code	✓	✓	✓	–	–	✓	–	✓
Mutated byte code	–	–	✓	–	✓	–	✓	✓
Summary report	✓	✓	✓	✓	✓	✓	✓	✓
Reduced test suite	–	–	–	–	–	–	–	✓
Documentation quality	Good	Insufficient	Sufficient	Sufficient	Insufficient	Insufficient	Insufficient	Good
Mutation operators								
Data types								
Primitives	✓	✓	✓	✓	✓	–	–	✓
Arrays & classes	✓	✓	✓	✓	✓	–	–	–
Pre-defined operators								
Assignment	✓	–	✓	✓	✓	–	✓	–
Arithmetic	✓	✓	✓	✓	✓	✓	✓	✓
Unary	✓	✓	✓	✓	✓	✓	✓	✓
Relational	✓	✓	✓	✓	✓	✓	✓	✓
Conditional	✓	✓	✓	✓	✓	–	✓	✓
Bit-level	✓	✓	✓	✓	–	–	✓	–
Java-specific								
Method signature	–	✓	✓	✓	–	✓	–	–
Keywords	✓	–	✓	–	–	–	✓	–
Object orientation								
Inheritance	–	✓	–	–	–	–	✓	✓
Polymorphism	–	✓	–	–	–	–	✓	✓

[1] * - The tool was not used in Students Survey
[2] ** - The tool works on Simple Java Classes

Regarding *Mutant inspection*, Bacterio and MuJava provide side-by-side windows showing the original code alongside the mutated versions. However, students were not always able to make those windows appear in Bacterio. All other tools provide information about the mutation operators applied to the lines of code (LoC), but Jumble provides this data only for live mutants.

As for the *Kill matrix* we observed that 4 tools do not render a kill matrix. Jumble provides a very coarse grained kill matrix showing for each mutation which is the test cases killing it. MuJava, Major, and Bacterio generate fine grained kill matrices displaying both test cases and test methods killing mutants.

As for the *Analysis runtime reduction* we observed that all the tools, implement at least one or use combinations of them. From the answers we gathered from authors, we were able to infer five possible values for this attribute. *Code coverage* indicates strategies that reduce the execution time of next iterations by prioritizing or excluding the execution of mutation operators on the basis of the code coverage achieved by a set of tests. *Test order* refers to mechanisms where the execution order or the exclusion of the test cases is determined before they are actually launched. *Mutant ranking* is a priority based mechanism that is applied to define the execution order of the mutant operators. *Infinite loops prediction* is a strategy to predetermine and to exclude the execution of the mutation operators that may produce infinite loops in the mutated code. In *Parallel execution*, mutation tools are executed in two or more JVMs running in parallel. The *Limited Mutants* mechanism executes only a limited number of mutants.

Regarding the *Equivalent mutant prevention* we observed that only 4 tools implement a mechanism for reducing the generation of equivalent mutants. PIT and Jumble provide a statistical one that executes operators having low probability to generate equivalent mutants. The code-based approach, implemented by Major and Javalanche, avoids the generation of equivalent mutants relying on code knowledge obtained through the AST analysis or by considering the code coverage reached by the test cases.

5.4.4 User-centric features. Regarding the *User interface*, we observed that almost all the tools provide a Command Line Interface (CLI) with exception of Bacterio providing only a Graphical User Interface (GUI). By default, MuJava comes as a tool that should be used through its GUI, but it provides muScript that is a CLI allowing direct access to key functionality provided by MuJava. Moreover, PIT, MuJava, and Jumble distribute plugins to extend well known Java IDEs like Eclipse or IntelliJ. As for the *Required inputs*, all the tools need test cases as input. Moreover, 5 tools over 8 require as input the Java byte code of the classes to be mutated, the remaining 3 tools work on the Java source code. PIT requires both source and byte code. Even if PIT applies mutations to the compiled code it needs source code for evaluating the code coverage reached by the test cases. Moreover, PIT is able to launch also test cases implemented in TestNG. Regarding the *Produced outputs*, all the tools produce a report summarizing the results of the tool execution. PIT and Major also produce a code coverage report showing the code executed by the test cases. Likewise, PIT and Major classify killed mutants, distinguishing between mutants that crash during execution, mutants that are killed by an assertion, and mutants that time out. Other tools only provide aggregate metrics such as the number of generated and killed mutants. Moreover, MuJava, Bacterio, Jester, and Major produce the mutated source code. Bacterio, PIT, Major, Javalanche, and Judy generate mutated byte code. Whether mutated source code or mutated byte code is preferable depends on the use case. For instance, generating source code mutants is important for mutation-based testing, which involves reasoning about the mutated code to develop new tests. In contrast, mutated byte code may be more efficient and sufficient for performing a mutation analysis for simply measuring test-suite efficacy. Bacterio also produces a reduced JUnit test suite.

As for the *Documentation quality* we observed that Bacterio and PIT provide *Good* documentation. However, even with good documentation, Bacterio was very difficult to use. The tool did not behave as described in the documentation and some students were not able to select or to execute the test cases. Major, and Jumble do not provide a clear (*sufficient*) documentation and students felt they would need more information, for instance, to help them with the installation process. MuJava, Javalanche, Jester and Judy need to improve the documentation provided. It is clearly not enough (*insufficient*). MuJava, Javalanche and Jester do not provide enough information to support the installation process. Judy was difficult to use and more information is needed to interpret final report. Also, Jester does not provide enough information to understand how the configuration file may be built/updated.

5.4.5 Mutation operators. In our analysis we identified 12 different types of mutation operators, corresponding to the 4 attributes reported in the framework:

Data types

(1) *Primitives:* mutation operators for variables, constants, and literals with a primitive data type.
(2) *Arrays & classes:* a dual to *Primitives* for array and class types.

Pre-defined operators

(3) *Assignment:* mutation operators that introduce, delete, or modify an assignment.
(4) *Arithmetic:* mutation operators that modify an arithmetic operator or its operands.
(5) *Unary:* mutation operators that modify a unary operator or its operand.
(6) *Relational:* mutation operators that modify a relational operator or its operands.
(7) *Conditional:* mutation operators that modify a conditional operator or its operands.
(8) *Bit-level:* mutation operators that modify a bit operation (incl. bit shift) or its operands.

Java-specific

(9) *Method signature:* mutation operators that delete a method call or that modify a method declaration or method call, e.g., by swapping arguments or formal parameters.
(10) *Keywords:* mutation operators that introduce, delete, or replace Java keywords such as `this`, `static`, `transient`, or `synchronized`.

Object orientation

(11) *Inheritance:* mutation operators that affect inheritance and dynamic polymorphism, including insertion and deletion of the keyword `super`, hidden variables, and overridden methods.
(12) *Polymorphism:* mutation operators that affect static polymorphism (incl. ad-hoc polymorphism), including insertion and deletion of overloaded methods and type-cast operators.

As Table 3 shows, MuJava and Major cover the most mutation types (10 out of 12), followed by PIT, Jumble, and Judy (each covering 9 out of 12). Jester covers the fewest mutation operator types (4 out of 12). From a different point of view, *Arithmetic*, *Unary*, and *Relational* mutation operators are implemented by all the tools, followed by *Primitive Data Types* and *Conditional*, that are supported by most of the tools. It is interesting to show that no tool provides mutation operators able to inject mutations related to the concurrent nature of Java. Also the *Inheritance* and *Polymorphism* mechanism mutation operators have a low support since they are implemented only by 3 tools.

6 CONSIDERATIONS FOR CHOOSING A MUTATION TOOL

We assert that the question of what is the best or most suitable mutation tool has no generic answer and depends on the concrete use case. Specifically, when researchers, educators, or practitioners select a mutation tool, their choice is motivated by different considerations.

To better understand what considerations are most important and to what extent our framework is sufficiently detailed, we created three questionnaires with identical questions for research[5], education[6], and practice[7]. These questionnaires were anonymous, based on our framework attributes, and included 21 specific questions plus 1 open-ended question for additional comments. We sent links to these questionnaires to contacts in academia and industry, whom we also asked to share them with other colleagues in the mutation analysis and mutation-based testing domain.

We received a total of 47 answers: 24 from researchers, 14 from educators, and 9 from practitioners. Based on the answers, this section outlines common and use-case-specific considerations. For simplicity, it refers to "important", "very important", and "mandatory" responses collectively as important considerations. It also shows how the proposed framework can aid in selecting a suitable tool for research, education, and practice by linking important considerations to *attributes* in Table 3.

6.1 Common considerations

It is desirable to select a mutation tool that is *actively maintained and evolving*. This increases the chances of using a state-of-the-art approach and timely resolving questions and issues. Another consideration is *compatibility* of a mutation tool, in particular w.r.t. supported testing frameworks and language features. A major challenge for comparing mutation tools is the lack of standardized descriptions and labels for *supported mutation operators*. Mutation tools may name the same mutation operator differently, they may use the same name for related yet distinct mutation operators, or they may apply the same mutation operator in different scopes. Consider Table 4, which demonstrates this challenge. Specifically, this table shows what mutants each of three mutation tools generate for the statement `return x+10;`. To produce this table, we applied three tools, MuJava, Major, and Jester (enabling all mutation operators they support) to the return statement. MuJava applied 15 mutation operators, Major 6, and Jester just 1. Moreover, MuJava and Major name related mutation operators differently, and Jester does not name them at all. Finally, *efficiency* is a general concern, but specific requirements may differ between use cases.

Based on the responses to our questionnaires, we observed the following. Out of 47 respondents:

- 35 consider active tool maintenance (*Release year* and *Release version*) important;
- 34 consider support for recent *Testing frameworks* important;
- 32 consider support for recent *Java versions* important.

PIT and Major satisfy all three considerations. Javalanche, Jester, and Bacterio are compatible with older versions of Java or JUnit. A related, cross-sectional concern is whether a tool works on sufficiently complex code (in Table 3, two asterisks next to the *Java version* indicate that a tool does not). The tools working on real projects are PIT, Major, Jumble, and Judy.

Furthermore, 44 out of 47 respondents consider a tool's capability for *Mutant inspection* important. In particular, out of 47 respondents:

- 39 consider mutated source code an important tool output (*Produced outputs*);
- 35 prefer source code as the input to the tool (*Required inputs*);
- 35 prefer mutations to be applied at the source-code level (*Mutation level*).

The tools fulfilling these requirements are MuJava, Major, and Jester. An interesting observation is that 38 out of 47 respondents do not consider mutated byte code an important tool output (i.e., "not important" or "do not care"). Additionally, 24 out of 47 respondents prefer a side-by-side visualization of the mutated and original code; MuJava and Bacterio are the only tools supporting this functionality.

[5]https://forms.gle/Vi56rip3FJrTKRJf9
[6]https://forms.gle/otUq72EbHkpDzUccA
[7]https://forms.gle/jKVt5RkreDzFcmdr6

Table 4. Mutant operators applied by MuJava, Major and Jester to statement `return x+10`.

MuJava	Major	Jester
AORB	**LVR**	return x+10; → return x+20;
x + 10 → x * 10	10 → 0	
x + 10 → x / 10	10 → -10	
x + 10 → x % 10	**AOR**	
x + 10 → x - 10	x + 10 → x % 10	
AOIU	x + 10 → x * 10	
x → -x	x + 10 → x - 10	
AOIS	x + 10 → x / 10	
x → ++x		
x → –x		
x → x++		
x → x–		
LOI		
x → ~ x		
SDL		
return x + 10; → return 0;		
VDL		
x + 10 → 10		
ODL		
x + 10 → x		
x + 10 → 10		
CDL		
x + 10 → x		

Finally, 45 out of 47 respondents rate a comprehensive *Documentation quality* as important, which is something that most tools lack. Only PIT and Bacterio provide a good documentation. Additionally, 39 out of 47 consider a standardized description of the supported *Mutation operators* important.

6.2 Research

Selecting a mutation tool for research purposes requires some unique considerations. For example, foundational research exploring the effectiveness of individual mutation operators and mutant selection strategies (e.g., [13, 20, 24, 44]), requires a high degree of *configurability* for selecting mutation operators. Furthermore, studying subsumption relationships and redundancy [1, 23] requires the *computation of a complete kill matrix*.

From the answers to our questionnaires, we observed the following. Out of 24 researchers:

- 22 consider a detailed summary report (*Produced outputs*) important;
- 19 consider producing a *Kill matrix* important;
- 18 prefer a high degree of configurability for *Mutation operator selection*.

Tools satisfying these requirements are MuJava, Major, and Bacterio. The most configurable tools are MuJava and Major, which support the selection of individual and classes of mutation operators.

Finally, a response to the open-ended question supports the notion of different use cases, suggesting that a complete kill matrix should be computed during *"execution in research mode"*.

6.3 Education

The selection of a mutation tool for educational purposes is less constrained by having access to a highly-configurable tool that implements state-of-the-art approaches or achieves a high degree of developer productivity. Likewise, the goal for this use case is not for students to overcome the difficulties of the installation process and a steep learning curve, but rather to *learn important concepts* related to mutation-based testing and understand its challenges and benefits. Therefore, the *ease of installation and use* and the existence of *supporting documentation* are relevant concerns.

Another relevant aspect is the presentation and *interpretability* of the produced artifacts. For example, a tool may only produce a summary of the generated mutants but no mutated source code for inspection. Likewise, a tool may provide information about why a mutant was (not) killed, whereas others may require additional tooling. As a result, a self-contained tool that works on small examples may be preferable to tools that work at scale but need to be fully integrated into a developer's workflow. Similarly, a GUI may be preferable to a command-line interface.

Based on the responses to our questionnaires, all educators prefer free/open-source tools, and many identify ease of installation as an important consideration. Additionally, out of 14 educators:

- 12 consider IDE integration, a tool-specific GUI, or both an important type of *User interface*;
- 9 prefer a side-by-side comparison of mutated and original code for *Mutant inspection*;
- 9 consider producing a *Kill matrix* important.

Javalanche, Jester, and Judy do not provide any type of GUI, and MuJava and Bacterio are the only tools that provide a side-by-side comparison. MuJava, Major, and Bacterio support the computation of a complete kill matrix.

6.4 Practice

Adopting a mutation tool in practice usually requires its *integration with the existing development environment*. For example, some tools only work with certain testing frameworks and Java versions. Another important aspect is *ease of interpretation*—that is, quickly understanding how a mutant was generated and how it can be killed. This is particularly important for developers if mutants serve as test goals. Likewise, *suppressing unproductive mutants* and *preventing equivalent mutants* are important to maximize developer productivity. Another consideration is the mutation level—that is, whether a tool mutates the source or byte code. While these details are often not clearly described in a tool's documentation, they are important when reasoning about mutant interpretability and workflow integration.

Based on the responses to our questionnaires, we observed the following. Out of 9 practitioners,

- 8 consider *Analysis runtime reduction* techniques important;
- 6 consider *Equivalent mutant prevention* techniques important;
- 6 consider integration into a development environment (*Build-tool integration*) important.

PIT, Major, Jumble, and Javalanche include techniques for the first two aspects, and PIT, Major, and Javalanche are the most suitable tools for integration—these can be integrated with traditional CI/CD environments such that they can be run from scripts.

Unlike in other use cases, 5 out of 9 practitioners prefer tools that support automated *Test selection* and coarse-grained *Mutation operator selection*. Major is the only tool that fulfills both requirements.

6.5 Discussion

To provide an aggregated view and contrast the answers between use cases (research, education, and practice), Tables 5–7 summarize the expressed importance of, and preferences for, individual aspects of mutation tools. All three tables highlight percentages above 70% (↑) and below 30% (↓) to draw attention to the most important aspects as well as conflicts between use cases.

Table 5. Importance of tool aspects when considering a mutation tool. All questionnaire responses are grouped into important vs. not important, and the table shows the percentage of responses that fall into the first group (see text for details). The numbers in parentheses give the total number of responses for each use case. Percentages above 70% and below 30% are highlighted.

Aspect	Research (24)	Education (14)	Practice (9)
Comprehensive documentation	**96%** ↑	**100%** ↑	**89%** ↑
Support for recent JUnit version	**71%** ↑	**71%** ↑	**78%** ↑
Details about applied mutation operators	**100%** ↑	**100%** ↑	67% -
Detailed summary report	**92%** ↑	**100%** ↑	56% -
Mutated source code as output	**83%** ↑	**93%** ↑	67% -
Updated recently	**75%** ↑	64% -	**89%** ↑
Standardized mutation operator descriptions	**96%** ↑	**86%** ↑	44% -
Support for recent Java version	67% -	64% -	**78%** ↑
Kill matrix as output	**79%** ↑	64% -	44% -
Build-tool integration	58% -	57% -	67% -
Equivalent mutant prevention	67% -	43% -	67% -
Run-time reduction techniques	54% -	**29%** ↓	**89%** ↑
Reduced test suite as output	42% -	**21%** ↓	44% -
Mutated byte code as output	**21%** ↓	**14%** ↓	**22%** ↓

Table 6. Preference for type of user interface (multiple-choice question) when considering a mutation tool. Three questionnaire responses did not indicate a preference and are excluded; the numbers in parentheses give the number of retained responses for each use case. Percentages above 70% and below 30% are highlighted.

User interface	Research (22)	Education (14)	Practice (8)
Command-line interface (CLI)	68% -	**21%** ↓	**88%** ↑
Graphical user interface (GUI)	32% -	50% -	**12%** ↓
Third-party IDE integration (IDE)	45% -	**79%** ↑	**88%** ↑

Table 7. Preference for the underlined choice when considering a mutation tool. Responses that did not indicate a preference for a given choice are excluded; the given percentages and corresponding fractions in parentheses are for retained responses. Percentages above 70% and below 30% are highlighted.

Choice	Research (24)	Education (14)	Practice (9)
Mutation level: source code vs. byte code	**95% (18/19)** ↑	**92% (11/12)** ↑	**100% (6/6)** ↑
Input to the tool: source code vs. byte code	**100% (20/20)** ↑	**100% (9/ 9)** ↑	**75% (6/8)** ↑
License: free/open-source vs. commercial	**100% (21/21)** ↑	**100% (14/14)** ↑	**75% (3/4)** ↑
Mutant inspection: lines of code vs. side-by-side	42% (8/19) -	31% (4/13) -	56% (5/9) -
Test selection: automated vs. manual	32% (6/19) -	40% (4/10) -	56% (5/9) -
Mutation operator selection: individual vs. groups	**78% (14/18)** ↑	**75% (9/12)** ↑	**29% (2/7)** ↓

Table 5 lists the most and least important aspects for each use case. This table groups all responses for each aspect into two categories: (1) important (grouping "important", "very important", and "mandatory" together) and (2) not important (grouping "not important" and "do not care" together). The table reports on the percentage of responses that fall into the first category. While a comprehensive documentation and support for a recent JUnit version is important for all use cases, the importance of many other aspects varies by use case. For example, a detailed summary report and a standardized description of mutation operators are important for research (96%) and education (86%), but not in practice (44%). Likewise, a selected tool being updated recently is important for research (75%) and in practice (89%), but to a lesser extent for education (64%). Moreover, some aspects are mostly important for a single use case. For example, support for recent Java versions is important in practice (78%), and producing a kill matrix is important for research (79%). Regarding the least important aspects, outputting mutated byte code is not important for any use case, and outputting a reduced test suite is not important for education. The importance of run-time reduction techniques shows a conflict: while it is not important for education, it is important for practice.

Table 6 shows the preferred type of user interface for each use case. This was the only multiple-choice question in the three questionnaires, and it included a "do not care" option. Two responses to the research questionnaire and one response to the practice questionnaire did not indicate a preference; we excluded these responses. The responses show that IDE integration is a preference for education (79%) and in practice (88%). In contrast, a command-line interface is not a preference for education (21%), but again a preference in practice (88%). While a command-line interface is also preferred for research (68%), the differences among the three options are not as striking for this use case. A (dedicated) graphical user interface is not a preference for any use case.

Table 7 summarizes the expressed preferences for one of two implementation choices for each use case. As before, we excluded "do not care" responses on a per-choice basis. For example, five responses to the research questionnaire, two responses to the education questionnaire, and three responses to the practice questionnaire did not express a preference for the choice of mutation level. This is important context: the table reports on the percentage and fraction of responses that do prefer the underlined choice, calculated over the number of responses that indeed expressed an opinion. For all three use cases, there is a strong preference for mutating source code, as opposed to byte code, as well as providing source code as the input to the mutation tool. Similarly, there is a preference for tools being free/open-source, though only 44% (4/9) of responses to the practice questionnaire expressed an opinion on this. The preference for selecting individual mutation operators (e.g., mutating `a+b` to `a-b`) as opposed to groups of mutation operators (e.g., mutating all arithmetic operators) shows a conflict: individual mutation operators are preferred for research (78%) and education (75%), but not in practice (29%). Preferences for mutant inspection and test selection show a similar conflict, but the differences are not as pronounced.

Overall, the responses to the three questionnaires show that there is consensus among the three use cases about the importance of many aspects of mutation tools. However, the responses also highlight some conflicts. These results, together with the summary of the surveyed tools' features (Table 3) allow researchers, educators, and practitioners to select a suitable mutation tool, based on a ranking of aspects that are most important to them. These results also allow developers of mutation tools to make informed decisions about what features to implement and how to improve their tools, based on their target audience.

7 CONCLUSIONS

This paper presents the results of a meta-analysis of existing comparisons of Java mutation tools, following a Rapid Review (RR) literature process. First, it proposes a comprehensive mutation tool comparison framework encompassing five dimensions, each with multiple attributes. Second, it reports on an application of the proposed framework to 8 state-of-the-art Java mutation tools, based on a literature survey, a tool-author survey, and a student survey. Finally, the paper reports on a survey of researchers, educators, and practitioners to understand which of the tool characteristics are most important for choosing a mutation tool for a particular use case. The responses indicate common as well as use-case-specific considerations, and the paper shows how the proposed framework can be used to identify the most suitable mutation tool for a given use case.

The findings of this paper can inform future work. First, some framework dimensions warrant a deeper analysis. For example, standardized mutation operator descriptions require a deeper analysis of the actual definition and implementation of the tools' mutation operators. Likewise, the outputs and summaries produced by the mutation tools may benefit from a finer-grained classification. Second, understanding whether and how mutation tools deal with specific classes of mutants (e.g., mutants that fail assertions, mutants that cause memory errors, mutants that time out, etc.), and how important these classifications are for different use cases is another avenue for future work. Third, it could be of interest to conduct follow-up surveys of researchers, educators, and practitioners to better understand their rationales behind the expressed preferences for tool characteristics. Finally, applying the proposed framework and process to describe mutation tools for other programming languages, such as C/C++, JavaScript, and Python would be valuable.

8 ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation grant CCF-1942055.

REFERENCES

[1] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing theoretical minimal sets of mutants. In *Proceedings of the International Conference on Software Testing, Verification and Validation.* 21–30.

[2] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering.* 402–411.

[3] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[4] Bruno Cartaxo, Gustavo Pinto, Baldoíno Fonseca, Márcio Ribeiro, Pedro Pinheiro, Maria Teresa Baldassarre, and Sérgio Soares. 2019. Software Engineering Research Community Viewpoints on Rapid Reviews. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 1–12. https://doi.org/10.1109/ESEM.2019.8870144

[5] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. In *Proceedings of the International Conference on Automated Software Engineering (ASE).*

[6] P. Chevalley. 2001. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proceedings Eighth Asia-Pacific Software Engineering Conference.* 267–270. https://doi.org/10.1109/APSEC.2001.991487

[7] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbr&#252;cken, Germany) *(ISSTA 2016).* ACM, New York, NY, USA, 449–452.

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

[9] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the Symposium on the Foundations of Software Engineering.* 416–419.

[10] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[11] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 19–30.

[12] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Does Choice of Mutation Tool Matter? *Software Quality Journal* 25, 3 (Sept. 2017), 871–920.

[13] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the limits of mutation reduction strategies. In *Proceedings of the International Conference on Software Engineering (ICSE).* 511–522.

[14] Megan M. Grime and George Wright. 2016. *Delphi Method.* American Cancer Society, 1–6. https://doi.org/10.1002/9781118445112.stat07879 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118445112.stat07879

[15] Sean Irvine, Tin Pavlinic, Len Trigg, John Cleary, Stuart Inglis, and Mark Utting. 2007. Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007* (09 2007).

[16] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sep. 2011), 649–678.

[17] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* San Jose, CA, USA, 433–436.

[18] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* 437–440.

[19] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE).* Hong Kong, 654–665.

[20] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* 284–294.

[21] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 612–615.

[22] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How Effective Are Mutation Testing Tools? An Empirical Analysis of Java Mutation Testing Tools with Manual Analysis and Real Faults. *Empirical Softw. Engg.* 23, 4 (Aug. 2018), 2426–2463.

[23] Bob Kurtz, Paul Ammann, Marcio E Delamaro, Jeff Offutt, and Lin Deng. 2014. Mutant subsumption graphs. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops.* 176–185.

[24] Bob Kurtz, Paul Ammann, Jeff Offutt, Marcio E. Delamaro, Mariet Kurtz, and Nida Gökce. 2016. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE).*

[25] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.

[26] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Trans. Softw. Eng.* 40, 1 (Jan. 2014), 23–42.

[27] Pratyusha Madiraju and Akbar Siami Namin. 2011. Paraμ – A Partial and Higher-Order Mutation Tool with Concurrency Operators. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops.* 351–356. https://doi.org/10.1109/ICSTW.2011.34

[28] András Márki and Birgitta Lindström. 2017. Mutation Tools for Java. In *Proceedings of the Symposium on Applied Computing* (Marrakech, Morocco) *(SAC '17).* ACM, New York, NY, USA, 1364–1415.

[29] P. R. Mateo and M. P. Usaola. 2012. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *2012 28th IEEE International Conference on Software Maintenance (ICSM).* 646–649.

[30] Ivan Moore. 2001. Jester - a JUnit test tester.

[31] A Jefferson Offutt. 1992. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 1, 1 (1992), 5–20.

[32] A. Jefferson Offutt and Roland H. Untch. 2001. *Mutation 2000: Uniting the Orthogonal.* Springer US, Boston, MA, 34–44.

[33] Jeff Offutt. 2011. A mutation carol: Past, present and future. *Information and Software Technology* 53, 10 (2011), 1098 – 1107. Special Section on Mutation Testing.

[34] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. 2014. HOMAJ: A Tool for Higher Order Mutation Testing in AspectJ and Java. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops.* 165–170. https://doi.org/10.1109/ICSTW.2014.19

[35] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering (ICSE).* 75–84.

[36] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275 – 378.

[37] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *Proceedings of the International Conference on Software Engineering (ICSE).* 910–921.

[38] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* (2021).

[39] Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. 2018. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In *Proceedings of the International Workshop on Mutation Analysis (Mutation).* 47–53.

[40] S. Rani, B. Suri, and S. K. Khatri. 2015. Experimental comparison of automated mutation testing tools for java. In *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions).* 1–6.

[41] Emily Reynen, Reid Robson, John Ivory, Jeremiah Hwee, Sharon E. Straus, Ba' Pham, and Andrea C. Tricco. 2018. A retrospective comparison of systematic reviews with same-topic rapid reviews. *Journal of Clinical Epidemiology* 96 (2018), 23 – 34.

[42] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09).* ACM, New York, NY, USA, 297–298.

[43] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14).* Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[44] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *Proceedings of the International Conference on Automated Software Engineering (ASE).* 92–102.