Humans Versus Mutants: A Controlled Experiment Analyzing the Types of Mutation Operators Human-Based Testing Tend to Miss

Navid Galt
Department of Software Engineering
George Mason University, Fairfax, VA, USA
Email: ngalt@masonlive.gmu.edu

Abstract—*This paper presents results from a controlled study that analyzes human-derived tests and determines what types of mutation operators are commonly missed. Mutation coverage is viewed as the gold standard practice in testing software and is superior in fault detection compared to other types of coverages. Mutation analysis has a multitude of subcategorical mutation operators. The goal of this study is to look at common types of mutations missed when programmers create requirement-driven human-based unit tests as well as determine whether there is a correlation between the programmers' experience and the number and types of mutations missed. It was discovered that Conditionals Boundary Mutator (CBM) was the most commonly missed mutation operator by programmers of all levels of experience. Furthermore, it was concluded that the more experience a programmer possesses, the more mutations their unit tests kill.*

1. Introduction

Software engineers are always looking for ways to reduce faults in production environments, yet they so easily fall into the trap of imprudently preparing for production releases by foregoing crucial steps in the development and testing phases.

Mutation analysis is a well-known fault detection technique that imposes a set of requirements on test cases. When used in the context of testing, this criterion helps produce higher coverage and quality test cases. Mutation analysis can be administered at different phases in the process of testing software. It's one of the most expensive types of software testing, but it is the most powerful and effective, yielding high code coverage results [1, 13]. Mutation testing is so expensive because of the large number of mutants that need to be compiled and executed on one or more test cases [12].

As explained later in this paper, mutation testing tools can supplement humans-based testing and expose faults that might not be identifiable by the human eye. With mutation analysis and proper testing habits, faults can be detected earlier in the testing phases of software and lead to a reduction in costs during integration and system testing, as well as post-deployment stages [11]. *Figure 1* portrays typical development stages and correlative costs of encountering faults at each specific phase.

In this paper, I look at human derived requirement-driven unit tests and use mutation analysis on those tests to examine any correlation between the type of mutation operator missed and the experience of the programmer. The research questions are presented as follows:

Research Question 1: What types of mutants are typically not killed in human-based testing?

To investigate this research question, a set of metrics and five mutation operators were relied on to calculate the number of mutants that have survived. These will be further elaborated on in sections three and four.

Research Question 2: Is there any correlation between the number and type of mutation operators that are missed and a programmer's level of experience?

This research question will be used to find a possible underlying reason why programmers miss certain mutation operators. A demographic survey, which was administered to each participant before this experiment, along with the results of the experiment will be reviewed and analyzed in conjunction to one another with the hopes of finding a correlation between the mutation operators missed and programmer's experience level. If no correlation is found, this will lead researchers to believe there is another underlying cause for mutations surviving human-based unit tests.

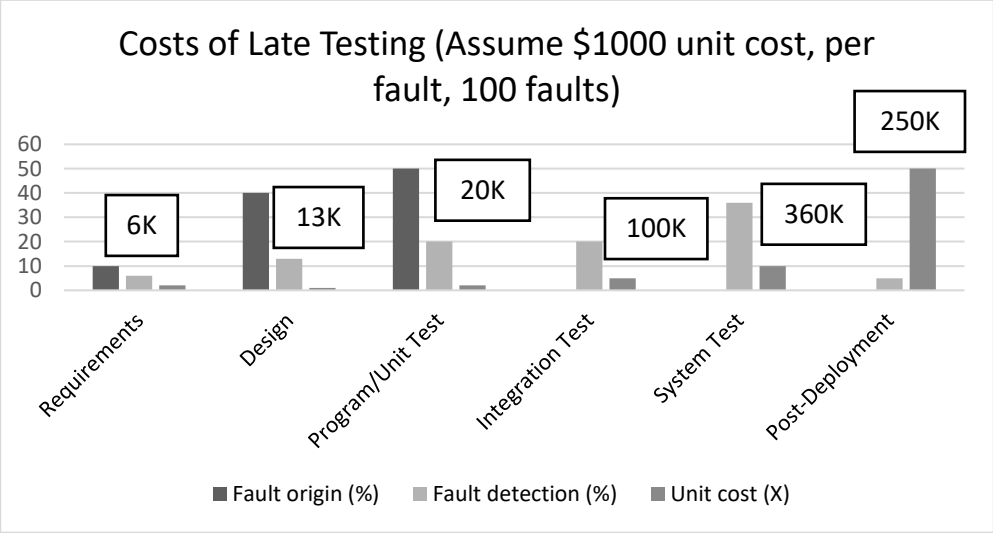

Figure 1: Costs of Late Testing [11]

The remainder of the paper is structured as follows: Section two furnishes a detailed view of mutation testing, its uses, and its effectiveness. Section three delineates the experimental protocol and procedure. Section four reports on the corresponding results, provides a thorough analysis of these results, and outlines construct, internal, and external threats to validity. Section five presents related work. Section six reviews the results, establishes future implications of the results, and concludes this paper.

2. Background - Mutation Testing

2.1 What are mutations and how are they used in testing?

Mutation analysis was originally introduced by Richard Lipton in 1971, in an unpublished class term paper "Fault Diagnosis of Computer Programs". However, it was not until the late 70's and early 80's when these ideas were outlined and consolidated into a major works by DeMillo, Lipton, Sayward, and Budd [14, 15, 16, 18].

Mutation analysis is used as a test criterion and measures the effectiveness of a testing technique based on changes that are systematically injected into programs under test. Each change inserted leads to a small syntactic variation of the original program, which is called a mutant [1]. According to Offutt [23], syntactic descriptions are used in mutation analysis to create tests. Offutt abstracts this, labeling it as a test criterion for "grammar-based artifacts". This interface serves as a platform on which mutation analysis implements. Theoretically, there are four different classifications of

grammar-based testing: program-based, integration, model-based, and input-based [23]. This study will specifically focus on program-based testing.

Being that mutation analysis is syntax-based, numerous mutation operators have been proposed for different programming languages and purposes [19, 20, 21, 22]. Typical mutations include replacing or adding operators or operands, negating or manipulating conditional statements, altering the values of constants and literals, commenting out a line, inserting exception statements, and many more. It is important to note that mutation analysis is a testing criterion, not a testing process. According to Offutt and Untch, "a testing criterion refers to a rule or collection of rules that imposes requirements on a set of test cases" [24]. To reiterate, mutations are based on a programming language's syntax and therefore are language specific. This paper focuses specifically on the mutation operators within the Java 8 framework.

To measure this testing criterion, researchers use what is known as a mutation score which quantifies the effectiveness of the tests under examination. This score is a ratio of mutants that have been killed by the tests over the total number of mutants generated. A mutant is said to be killed if the tests under analysis are able to discern the output of the mutant from that of the original program [23, 24]. The goal of mutation testing is to find test data to kill all mutants [25]. A score of 1.00 indicates that all mutants have been killed. In the below equation, MS refers to mutation score, P refers to the program, T refers to the test cases, M represents the total number of mutants, and K represents the number of killed mutants.

$$MS(P,T) = \frac{K}{M}$$

Most tools and up-to-date formulas include redundant or equivalent mutants in their calculation, which in the above formula would be subtracted from M. This was excluded for simplicity reasons and because equivalent or redundant mutants are beyond the scope of this paper.

As outlined by Offutt and Untch [24], in a manual process, after using mutation operators, the tester must use his discretion to determine whether the output of the program on each test case is correct or incorrect. Based on that, the tester either continues the process or stops it to rectify the fault found. A mutation testing tool adds the benefit of automatically sifting and fashioning an intuitive interface. This allows testers and developers to analyze the results from the unit tests and determine mutation coverage (the ratio of reached and killed mutations) of the asssociated unit tests.

2.2 How effective is mutation testing?

As the reader, you might be wondering how exactly does it ensure actual faults will be exposed and how effective is this criterion?

Although faults used in mutation analysis are artificial, they sometimes mimic actual faults a system might encounter. According to Geist et al. [25], if a piece of software contains a fault, mutation analysis will assist by creating a set of mutants that will only be killed by a test case that also detects a bona fide fault. In other terms, after a mutant is generated, if the unit test set does not kill that mutant then most likely if an actual fault were to occur in the system this fault would go unnoticed by the unit tests. This is very useful for developers and testers. As humans there are occasions and paths that can go unnoticed; these can be attributed to typical human error. With this concept in place, development and testing teams can ensure that all paths are covered helping to detect and reduce failures possible failures.

Mutation testing is known as the golden standard, at no small cost though. Even trivial programs can produce an exceedingly copious number of mutants, of course reliant on the number of mutation operators chosen. Mutation analysis can be exceedingly time-consuming and quite expensive computationally, especially in comparison to structural code coverage criteria [26]. However, mutation analysis is superior in fault detection compared to other types of coverage. Frankl et al. states that mutation testing is superior to data flow coverage criteria [27] and Walsh noted in his Ph.D. Thesis in 1985 that mutation testing is more powerful than statement or branch coverage [28].

3. Experimental Protocol

3.1 Participants

This study employed 7 subjects. All participants attend George Mason University, with three of them in their graduate studies and four of them in their undergraduate studies with a Computer Science or Software Engineering focus. Six of the participants were male and one female. The ages of the participants ranged vastly. Four participants were in the 20-26-year range, two of them in the 27-33-year range, and one of them in the 34-40-year range. Six of the participants had more than four years of total programming experience, with a split range of industrial work experience; three participants had between zero and one year, two participants had between one and two years, and two participants had more than four years of experience. Most participants chose Java as their most fluent language, but all participants were capable of writing JUnit tests. Out of all the participants only 14% always write unit tests for programs they create. The other 86% create them occasionally. What stood out is that over 90% of participants do not use some sort of model, modeling design tool, or code coverage criteria to derive and analyze their unit tests.

3.2 Variables

3.2.1 Dependent Variables

The first dependent variable will answer *Research Question 1* and the second along with the third dependent variables will answer *Research Question 2*. Experimenters will use mutation operators and measurement metrics outlined in section 4.4.1 and 4.5, respectively, to guide the measurement of these dependent variables:

1. Average number of mutations that survived in comparison to the total number of mutations generated for each mutation operator.
2. Based on level of programmer's experience which mutation operator was missed the most.
3. Based on level of programmer's experience and metric 2, which group had the highest quality of tests.

3.2.2 Independent Variables

In order to determine the programmer's level of experience, two factors were considered. First, the total number of years in which the programmer had experience programming. Second the number of years the programmer had worked in an industrial setting. The below classifications partition the level of programmer's experience into three categories.

1. Novice: Less than three years of general programming experience and less than two years of professional experience.
2. Intermediate: Three or more years of general programming experience and less than two years of professional experience.
3. Expert: Three or more years of general programming experience and two or more years of professional experience.

3.3 Experimental Hypothesis

I hypothesize that…

- With human-based testing, humans are more likely to kill some types of mutants.
- Lower experience level will show a positive correlation with the number of mutations that are typically not killed.

3.4 Experimental Material

3.4.1 Demographic Survey

Google Forms was chosen to administer this survey. Within the survey, participants were asked a series of questions, which were then utilized by experimenters to get a general idea of the demographics of the participants as well as background, skills, and experience. Questions inquired about participants age, gender, total years of programming experience, total years of professional work experience, participants' most fluent programming language, and other questions regarding test creation. This survey was used to employ our independent variable and partition graphical results properly.

3.4.2 Comprehension Survey

Google Forms was also chosen to administer this survey. Due to this study being administered remotely, this survey was used to indicate whether participants fully grasped the concept and functionality of the program which they were to create unit tests for. This was done to help alleviate any threat to internal validity by the participants not understanding the program and therefore, skewing the results.

3.4.3 Program Under Test

Researchers developed a program simulating a book library for which participants would develop JUnit tests for. This program consisted of three classes: Book.java housing a Book object, LibraryActions.java which creates a library of books and houses methods to access that library, and LibraryTests.java that functioned as a skeleton class with mock data that participants were tasked to complete. Book.java had a series of private fields along with getter and setter methods to access those fields. LibraryActions.java constructed an ArrayList of books that could be stored in the library and had a series of manipulation methods to insert, remove, and access those books. Overall, the program was around 300 lines of code. A more thorough list of methods in these classes are listed in Appendix A, following Section 5 on page 16.

3.4.4 PIT Mutation Testing Tool

For this experiment, an eclipse plug-in for PIT called Pitclipse was used. This plug-in uses the PIT mutation tool as its backend utility. PIT applies a configurable set of mutation operators to the byte code generated by compiling your code. PIT defines five default mutation operators, which are used in this experiment, that will mutate the bytecode in various ways. These default methods are…

1. **Conditionals Boundary Mutator (CBM):** This mutator replaces the relational operators <, <=, >, >= with their boundary counterpart < → <=, <= → <, > → >=, >= →>.
2. **Void Method Call Mutator (VMCM):** The void method call mutator removes calls to methods that're of type void.
3. **Return Values Mutator (RVM):** This mutator mutates the return values of method calls. Depending on the return type of the method another mutation is used. (i.e. for type Boolean PIT replaces the unmutated return value true with false and replace the unmutated return value false with true. For type int byte short, if the unmutated return value is 0 return 1, otherwise mutate to return value 0. For type long, PIT replaces the unmutated return value x

with the result of x+1. For type float and double, PIT replaces the unmutated return value x with the result of -(x+1.0) if x is not NAN and replace NAN with 0. For type Object, PIT replaces non-null return values with null and throws a java.lang.RuntimeException if the unmutated method would return null.

4. **Math Mutator (MM):** This mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. (i.e. + → -, - → +, * → /, / → *, % → *, & → |, | → &, ^ → &, << → >>, >> → <<, >>> → <<).
5. **Negate Conditionals Mutator (NCM):** will mutate all conditionals. == → !=, != → ==, <= → >, >= → <, < → >=, > → <=.

Further information on the selection of mutation operators offered by this tool is available on PIT's website [10].

Based on these operators, PIT generates a coverage report showing users percentages of line coverage, mutation coverage, the number of mutations that survived. And mutations that have been killed.

This tool was chosen for several reasons. First, it was the most up to date code base and actively developed and supported which allowed for up-to-date fault remediation, and recent frequently asked questions (FAQs) with answers and documentation. Second, its ease of use; Pitclipse was very simple to install and plugin to Eclipse, allowing researchers to spend less time on the proper functioning of the mutation tool and more time conducting the experiment. Several other tools were eliminated from consideration after being experimented with. All of these tools had a more difficult installation and running process which began to interfere with the time allotted to this experiment. Last, PIT accounts for equivalent and redundant mutants by not generating mutants with those characteristics. Large numbers of redundant mutants can inflate and provide an inaccurate mutation score [17].

3.5 Measurement Metrics

Two metrics were created to measure our dependent variables for this experiment. Both of these metrics subsume the mutation operators mentioned in the previous section in their derivation.

Metric 1 - Survived Mutation Score (SMS): The first metric is used to measure the first dependent variable. Each sub-metric is a variation of a typical mutation score, defined by Ammann et al. as "the fraction of mutants that are killed by a test set" [17]. The metric used here will slightly augment the standard definition and provide the ratio of survived mutations over the total mutations generated for each mutation operator.

The formula below is designated to measure the SMS for each of the five mutation operators; CBM, VMCM, RVM, MM, NCM. S is the number of survived mutants, while T serves as the total mutants discovered, and m represents the mutation operator type, all of these were identified using PIT.

$$SMS = \frac{S_m}{T_m}$$

Metric 2 – Weighted Average (WA): The second metric is used to measure the second and third dependent variable. This metric is calculated using a weighted average method. Weighted average is a category of arithmetic mean in which some elements of the data set carry more importance than others. In other words, each value to be averaged is assigned a certain weight. In mathematics and statistics, you calculate weighted average by multiplying each value in the set by its weight, then you

add up the products and divide the products' sum by the sum of all weights. For this experiment, this metric offered a more meaningful insight into the actual effectiveness of the tests developed by each participant. Among the mutation operators, line coverage calculated by PIT was also included.

This formula portrays how the effectiveness of the unit tests, X, created by each participant, p, was calculated. Where w is the weight of the i^{th} test (weight of the mutation operator) and x is the percentage of successful outcomes for the i^{th} test (also known as the mutation score). The sum of all w is assumed to be one.

$$\langle X \rangle_p = \frac{1}{n} \sum_{i}^{N} w_i x_i$$

This metric is important to normalize the accuracy of *SMS* because the quantity of mutations per test could be different and as such, unweighted test results themselves might not be fully indicative of a developer's competency. If more than one type of test result is combined into a score, a greater insight into test effectiveness can be gained. This metric permits us to rank participants based on various test outcomes, both mutation and line. This can be extended to additional levels of testing. It is important to note, that the weight of the i^{th} test, w_i, in the aforementioned formula, refers to each mutation operator in the calculation of this weighted average and was calculated based on O, the number of mutants that were generated for that mutation operator over M, the total number of mutants generated.

$$w_i = \frac{O}{M}$$

The weight is directly proportional to the larger number of mutations generated for that mutation operator.

3.6 Experimental Tasks and Procedure

This study was a controlled lab experiment. Each participant was administered the same program from which they created unit tests from.

Each participant was first tasked with completing a pre-experiment demographic survey to gauge the different factors that might play a role in the quality of the tests they created. After that, each participant was asked to review the book library program.

After reviewing the program, participants were given 10 minutes to complete a comprehension survey. This was used to ensure outside factors (i.e. programmers' comprehension) did not have an influence on the programmer's ability to derive unit tests for LibraryActions.java. If any ambiguity was expressed or observed the experiment administrator would help mitigate them. The participants were then given 45 minutes to create a set of JUnit tests. Time limits were enforced in order to maintain consistency and reduce the possibility of an internal validity threat. Participants were not allowed to exceed the time limit allotted. Mock data was provided in the form of Book objects. Participants were allowed to follow whatever pattern of creation they were comfortable with.

After the subjects completed the unit test portion, the experimenter used PIT, to see which mutation operators were killed by the human-designed tests and which ones survived. From there, an analysis was done using aforementioned formulas to gather metrics which were then used to try and find a correlation between the results of the number and type of mutants that survived and the level of

experience of the programmer. These metrics were also used to determine if there was a certain or common type of mutant that consistently survived.

4. Results

4.1 Overview

This section discusses the results of the statistical tests conducted in order to evaluate hypotheses defined in Section 3.3. The SMS formula was used to synthesize *Table 1*. *Figure 2* indicates which category of mutants are less likely to be killed by human-based test creation. *Figure 3* conveys how experience levels correlate to the number and type of mutation operators that survived. The WA formula was used to synthesize *Figure 4* which conveys how experience level correlates to test quality. An additional set of measurements will be introduced to discuss the statistical significance, portrayed by *Table 2* and graphically in *Figure 5.*

4.2 Pattern Measurement

As mentioned in Section 3.5, two formulas were used to measure the dependent variables and prove the hypothesis. Shown below, *Table 1* exhibits the mutation operators by category and reveals the highest missed mutants among the complete subject group. The first column fashions each mutation operator, the second is the average of mutants that survived out of the total number of participants, and the third column is the total number of mutants generated for each mutation operator by PIT in the book library program. Out of all participants, CBM has an average of .43 survived mutants out of 1 total generated mutants, VMCM has an average of .57 survived mutants out of 3 total generated mutants, RVM has an average of 6.14 survived mutants out of 24 total generated mutants, MM has an average of .43 survived mutants our of 2 total generated mutants, and NCM has an average of 3.57 survived mutants out of 26 total generated mutants.

Mutation Operator	Average Mutants Survived	Total Mutants Generated Per Operator
CBM	.43	1
VMCM	.57	3
RVM	6.14	24
MM	.43	2
NCM	3.57	26

Table 1: Average Mutants that Survived per Mutation Operator

To truly understand which mutation operator was more commonly missed by human-based testing, we need to apply the SMS metric delineated in Section 3.5. In *Figure 2*, the SMS formula was applied to determine which mutation operator was most commonly missed. The x-axis lists each mutation operator and the y-axis lists the SMS percentages. Before observing this, it's important to note that out of all the mutations generated by PIT, the RVM and NCM mutation operators had the greatest number of total mutants generated. Although a larger amount of RVM and NCM mutants were generated and survived, when SMS is applied it can be observed that CBM is the most likely mutation operator to be missed among all participants with an average SMS of 42.86%. The second most common mutation operator missed was RVM with a SMS of 25.60%. Following those mutation operators were MM, VMCM, and NCM with a SMS of 21.43%, 19.05%, and 13.74%, respectively.

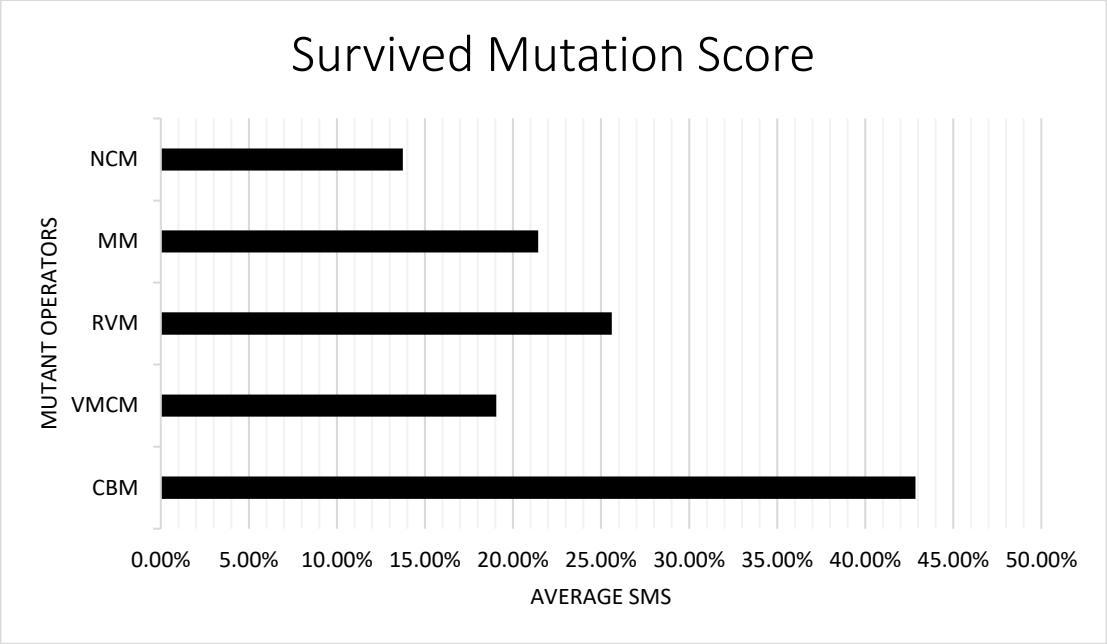

Figure 2: Survived Mutation Score (SMS) per Mutation Operator

To relate this data to our hypothesis, we charted the data collected in *Figure 3* based on the programming experience of the participants. From here, we could observe, on average, which mutation operator was missed the most by each group of programmers. *Figure 3 is a* three-dimensional graph. With the y-axis listing the number of mutations missed, the x-axis lists the partitions of programming experience level and the z-axis lists the five mutation operators examined in this experiment. For the CBM operator the Novice group had 0 survived mutants, Intermediate had 1, and Expert had 0.5. For the VMCM operator the Novice group had 1 survived mutant, Intermediate had 1, and Expert had 0.5. For the RVM operator the Novice group had 13 survived mutants, Intermediate had 6, and Expert had 3. For the MM operator the Novice group had 1 survived mutant, Intermediate had 1, and Expert had 0. For the NCM operator the Novice group had 6 survived mutants, Intermediate had 4, and Expert had 2.

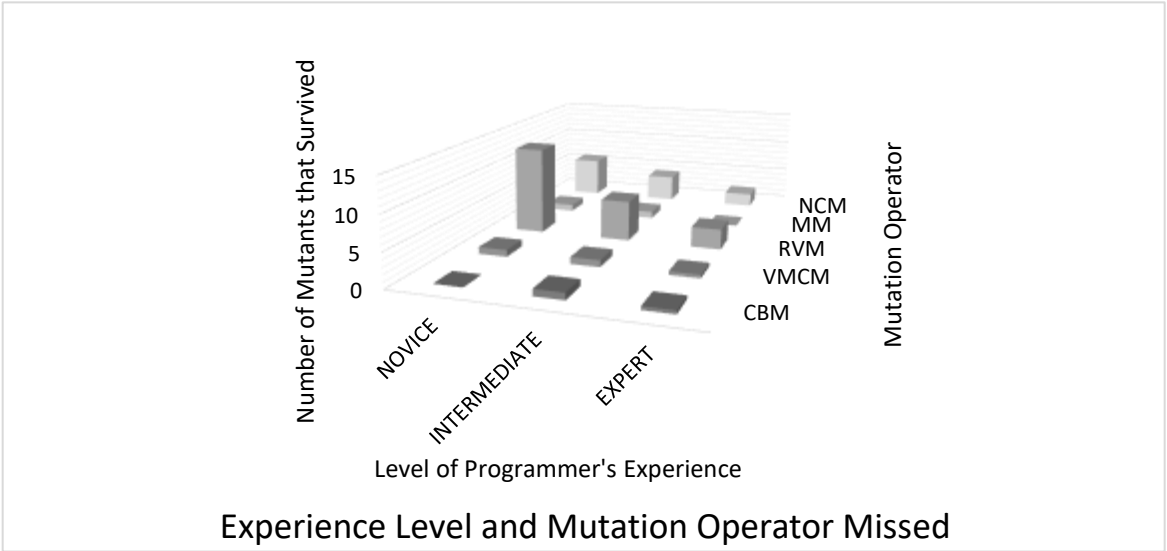

Figure 3: Average Number of Mutants that Survived per Mutation Operator per Programmers' Experience

To further analyze these results, metric 2 was used to determine the ranking of effectiveness in each group of programmers. In *Figure 4*, it can be observed that on average the Novice groups test effectiveness was around 63%, Intermediate 78%, and Expert 87%. The higher the effectiveness percentage, the more effective the test suite create by the participant was.

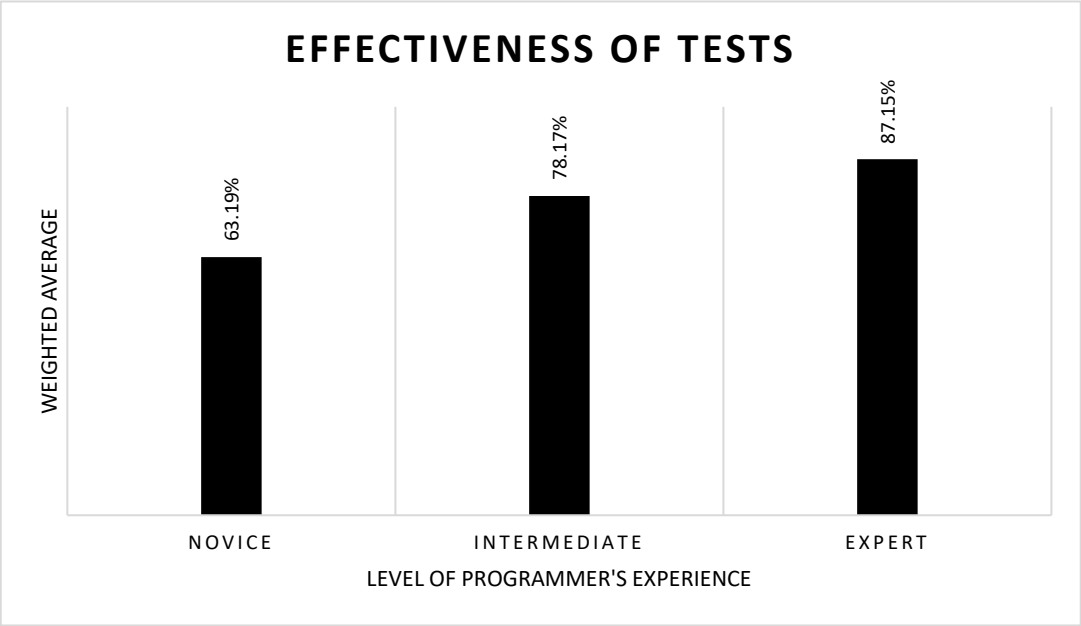

Figure 4: Effectiveness of JUnit Tests per Programmers' Experience

Furthermore, standard deviation was used to measure how far the data gathered was spread apart per experience level group. Standard deviation expresses the amount of variation existent in the mean. In other words, it represents a "typical" deviation from the mean. It is a popular measure of variability because it returns the original units of measure of the data set [34]. In addition to expressing the variability of a population, the standard deviation is commonly used to measure confidence in statistical conclusions. Effects that fall much farther than two standard deviations away from what would have been expected are considered statistically significant—normal random error or variation in the measurements is in this way distinguished from likely genuine effects or associations. A population standard deviation formula is given as follows:

$$\sigma x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

The standard deviation, σx, equates to the square root of the summation of x_i, each of the values of the data, minus $\overline{x}$, the mean or average of the data, squared. n serves as the upper bound for the data (the participants), and 1 the lower limit. (i.e. $x_1, x_2, x_3, x_4, \ldots x_n$).

We use the mean of effectiveness for each programming level and the standard deviation from the weighted averages of each programmer in all three programming levels to devise a bell curve. A bell curve is used to more accurately evaluate participant's performances and represent the data gathered. A bell curve, also known as normal distribution curve, is used to represent how data from a process is distributed and is defined by the mean, and the standard deviation. The center of the graph is the mean, and the height and width or spread of the graph are determined by the standard deviation.

When the standard deviation is small, the curve will be tall and narrow in spread. When the standard deviation is large, the curve will be short and wide in spread. We can determine how anomalous a data point is based on how many standard deviations it is from the mean. When you have a dataset that is normally distributed, your bell curve will follow the below rules [34]:

- The center of the bell curve is the mean of the data point (also the highest point in the bell curve).
- 68% of data is within ± 1 standard deviations from the mean
- 95% of data is within ± 2 standard deviations from the mean
- 99.7% of data is within ± 3 standard deviations from the mean

Using the standard deviation formula, we calculate the standard deviation per experience level. These calculations are depicted in *Table 2*. Novice has a standard deviation of 0. Intermediate has a standard deviation of 16 and Expert has a standard deviation of 3. Due to only having one classified participant, this measurement is not applicable for the *Novice* group. For the *Intermediate* group 50% of participants are within one standard deviation from the mean (62-94) and the other 50% are within 2 standard deviations (46-110). For the *Expert* group, 100% of participants are within 1 standard deviation from the mean (84-90).

Programmer Experience Level	Standard Deviation	One Standard Deviation Range	2*Standard Deviation Range	3*Standard Deviation Range
Novice	0	*N/A*	*N/A*	*N/A*
Intermediate	16	62-94	46-110	30-126
Expert	3	84-90	81-93	78-96

Table 2: Standard Deviation per Programmers' Experience

Figure 5 depicts 2 bell curves in one graph. The *Novice* group's bell curve is not depicted because the calculated standard deviation is 0 and therefore no bell curve is observed. The *Intermediate* group curve has a large standard deviation. The *Expert* group curve has a small standard deviation.

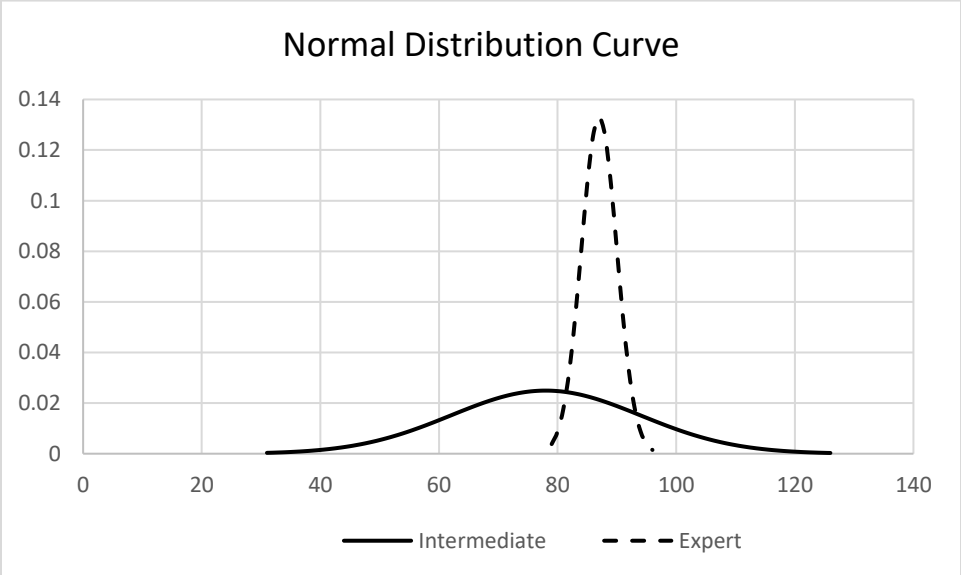

Figure 5: Normal Distribution Curve Per Programmers' Experience

From this graph, we can form a null hypothesis H_o: the data does not follow the specified normal distribution. Performing the Kolmogorov-Smirnov (K-S) test, we evaluate the data and either accept

or reject the null hypothesis. A K-S test can be used to compare actual data to normal distribution. This test takes the cumulative properties of the values in the data and compares them with the cumulative probabilities in a theoretical normal distribution. There are several criteria which the K-S test operates on. A critical value of D_α is used to measure the level of significance which is what α represents. For this test, we use the most common significance level, $\alpha = 0.05$. Then, for each group the critical value was calculated based on the sample size for each *Programming Level* group [35]. The *Intermediate* group we will use a D_α value of 0.62394, for *Expert* we use a D_α value of 0.84189. In order to reject null hypothesis this test must calculate a value greater than the critical value $D > D_\alpha$, else we accept it. Microsoft Excel was used to perform these calculations. For *Intermediate* we the K-S test statistic value is 0.81645 and the *Expert* group statistic value is 0.26025. This means that for the *Intermediate* group we can reject null hypothesis (0.81645 > 0.62394), however in the *Expert* group we accept null hypothesis (0.26025 < 0.84189). Furthermore, a two sample K-S test can be used to test whether two samples come from the same distribution. This can be conducted in a pairwise fashion for all permutations of experience level in order to accept or reject null hypothesis H_o: Experience level will have a similar normal distribution defined by mean and standard deviation of test effectiveness. This would show that the SMS indeed serves as a good discriminator of experience level.

4.3 Discussion

In *Figure 2*, it was observed that based on the mutation operator with the highest SMS the CBM operator was the most likely operator to be missed by programmers. Following that were RVM, MM, VMCM, and NCM, respectively. In *Figure 3*, a correlation was observed between experience level and number of mutants that survived. For the most part, the higher classification of experience the less mutants that survived from the participants' human derived unit tests. In *Figure 4*, according to metric 2 it was observed that the *Expert* level programmers created more effective tests. In *Table 2* and *Figure 5* it can be determined that both the *Novice* and *Expert* groups have low standard deviations while the *Intermediate* group has a slightly higher standard deviation. There was a low number of subjects for each class and given this concern we can surmise that the findings in this experiment were statistically insignificant. However, the metrics and analysis methodology is a sound framework that can scale infinitely large if performing such an experiment on a large sample is desired.

4.4 Threats to Validity

Throughout this experiment, numerous threats to validity were observed. Scientists do their best to mitigate these threats, however, in occasional circumstances this might not be feasible. It is incumbent of the experimenters to make these threats known and address them to the best of their abilities. This section describes these threats, primarily on how each threat was addressed, but also cases where they were not adequately addressed. The threats are described according to the classification of Easterbrook et al. [33].

4.4.1 Construct Validity

Construct validity focuses on whether the theoretical constructs are interpreted and measured correctly. Problems with construct validity occur when the measured variables don't correspond to the intended meanings of the theoretical terms.

A threat to construct validity that is ubiquitous among mutation analysis experiments is the assumption that mutants are not a valid substitution for real faults, however, as mentioned earlier,

there is a considerable amount of empirical evidence indicating that mutants are a valid substitute for real faults [4, 5, 6, 25].

4.4.2 Internal Validity

Internal validity focuses on the study design, and particularly whether the results really do follow from the data. Typical mistakes include the failure to handle confounding variables properly, and misuse of statistical analysis.

Adverse results can occur during experimentation from disparate Java programs. To control this and avoid any possible internal threats, the same Java program was used across all subject groups.

Due to the limited sample size for this study and the targeted community, inclusion criteria [30] was not used to fully determine the experience of the programmer in this defined language, Java. This knowledge was partially measured by the demographic survey where the participants most fluent language was inquired about, however no control was put in place during or after the experiment. As a result, the outcomes observed in this experiment could've indirectly been affected by the participant's Java programming language proficiency.

Redundant and equivalent mutants tamper with mutation score: In the empirical study [1] conducted by Just and Schweiggert, they show how the inclusion of redundant mutants leads to an inaccurate mutation score. To control this, the PIT Mutation tool was used to conduct the mutation analysis portion. PIT abstains from generating certain types of equivalent mutants by not generating mutations for lines that contain a call to common logging frameworks [10]. The theoretically adverse and limited mutation operators the PIT mutation tool offers could also present a possible threat to internal validity. Each mutation tool researched generated different amounts and types of mutation operators based on a theoretical framework. Disparate or additional operators may affect both the number and the ratio of survived and generated mutants. According to Wong and Mathur [31] and Offutt et al. [32] a collection of mutation operators that insert unary operators and that modify unary and binary operators will be most effective. However, these mutation operators were not a part of the default methods offered by PIT and as a result were not used in this study. This results in another threat to internal validity not controlled for.

Time constraints affected this experiment in many facets. As mentioned throughout this paper, experimenters were under a time constraint and expediting procedures became a norm at the cost of a quality experiment. Participants could have also been influenced directly; an average time limit was established for the creation of the JUnit tests based on the time it took the experimenter to complete. Indubitably, this window was elongated to allow enough time for participants, however this time period was not catered to experience level and therefore could've negatively affected the quality and coverage of tests produced by participants.

As mentioned above, quality was sacrificed due to time constraints. A large portion of the experiments were conducted remotely, this is a threat to internal validity and could have affected how participants interpreted the instructions and program. To address this issue, a program comprehension survey was used to identify any confusion, if there was any, and notify the researcher so any misunderstandings could be mitigated. In retrospect, a way to have further mitigated this threat would've been to thoroughly comment the code in the library program.

4.4.3 External Validity

External validity focuses on whether claims for the generality of the results are justified. Often, this depends on the nature of the sampling used in a study. For example, if Jane's experiment is conducted with students as her subjects, it might be hard to convince people that the results would apply to practitioners in general.

A threat to external validity caused by the time crunch was the size of the program, which was considerably small, just under 300 lines of code, but was necessary to stay within a reasonable timeframe for the participant. These results are specific to this program and might not be applicable to programs of other purpose or size.

Another external threat was the low number of participants that were gathered to conduct the experiment. This was due to an inability to properly market the study in the time given. Therefore, these results might not be scalable to larger experimental groups. There was an uneven distribution of participants per subject group. This could've affected the accuracy of the results.

Another possible threat to external validity is that although all participants came from different backgrounds and had different skill levels, all students are attending George Mason University. This has the potential to bias results, because all students come from a similar educational background. Measures were not put in place to control this threat. This was partly due to the time constraint which was given for the completion of this experiment and paper.

5. Related Work

A search in relevant literature showed that mutation analysis has not been used for comparative measures to human-based test development. However, several studies examine mutation analysis to draw their own conclusions about the benefits of its use.

In Section 2.2, we delineate these related works as it provides sustenance in proving that mutation analysis is an effective test criterion framework. Geist et al. [25], found that mutation analysis when used as an automated test generator helps to reduce system failures in the design of fault-tolerant software systems and improves system reliability. Frankl et al. found that among most test subjects, mutation testing was more effective in comparison to data flow coverage criteria [27]. Walsh also noted in his Ph.D. thesis that mutation testing is more powerful than statement or branch coverage [28]. In regard to the effectiveness of mutation analysis, its benefits are pronounced. To reiterate this study focuses on program-level mutations. Another interesting area of application is class level mutations. In their paper, Kim et al. [29], analyzes class level mutations comparing effectiveness of different test creation strategies in terms of fault-finding ability.

As a side note, it is worthy to state that initially, MuJava was chosen as the mutation analysis tool to be used for this experiment, due to its comprehensive coverage. As for related work, a side by side comparison provided by Rani et al., discovered that the MuJava tool offers more coverage compared to other tools including PIT [3]. However, as mentioned earlier in Section 3.4.4, the experimenters ran into several issues setting up and conducting the experiment with the MuJava and other tools, and as a result diverted to PIT.

6. Conclusion and Implications for Future Research

To recap, this paper investigated and sought to analyze what types of mutation operators are commonly missed in human-based requirement driven unit tests. In this experiment, the default mutation operators that PIT offered were used to simplify the mutation analysis portion. Focusing on these five default mutation operators, this paper demonstrates that the CBM operator was more likely

to survive in tests humans developed from requirements when no test modeling was involved. This research also found that more experienced programming participants yielded less survived mutants and derived more effective tests.

Due to time constraints, a large portion of possible mutation operators were not used. In future work, it would be interesting if experimenters replicated the study and added other program-level mutation operators. According to PIT's website [10], there are over 29 different and available mutation operators. Offutt et al. [23], defines eleven effective operators with the Java programming language. As mentioned earlier, there are different phases and levels of testing; class, integration, and systems – all which mutation analysis can be applied to. Specifically, mutation analysis is also applicable in the context of integration or system testing (e.g., [7, 8, 9]). Future studies could look at these levels and understand what type of mutation operators are typically missed in these stages. To further build on these results, more experimentation needs to be conducted. Moreover, as mentioned in Section 4.3, scaling this experiment to a larger participant group is possible with this framework and is highly encouraged. This approach also serves as the basis for an automated classification system that could lead to interesting new advances in Machine Learning and Expert Systems for industrial software development. If industrial programmers understand what types of mutations their software teams typically miss, they'll be able to closely monitor these conceivable faults to ensure that proper testing is administered and executed, proliferating fault detection. This would have profound benefits in safety and mission-critical systems, where a fault can mean the difference between bankruptcy and furthermore, life and death.

The PIT mutation framework and documentation is publicly available at:
http://pitest.org/

Appendix A

- Book.java
  - Fields:
    - String bookTitle, serialNumber, authorName, publisher
    - int publishedYear, bookQuantity
  - Methods: //getter and setter methods used to retrieve and set above values in a book object
    - getBookTitle(), setBookTitle(), getSerialNumber(), setSerialNumber(), getAuthorName(), setAuthorName(), getPublisher(), setPublisher(), getBookQuantity(), setBookQuantity(), getPublishedDate(), setPublishedDate()
- LibraryActions.java
  - Fields:
    - Arraylist<Book> books //data structure used to house library of books
  - Methods:
    - insertSingleBook() //takes in a book object, if the book doesn't exist in the library this method inserts it, if it exists this method checks in the book. In both instances it returns the book's serial number.
    - isBookAvailable() //takes in a book object, checks to see if there is an available copy of the book, if there is it returns true, if there's not it returns false.
    - checkOutBook() //takes in a book object, reduces the quantity property by one, and returns the book, if there is zero quantity it returns null.
    - checkInBook() //takes in a book object, increases the quantity property by one, and returns true if the book exists in the library, else it returns false.
    - searchForBookByAuthorsName() //takes an author's name, throws NPE if name is null, if a book exists with author's name it returns that book, else it returns null.
    - searchForBookByBookTitle() //takes in a book title, if title is null NPE is thrown, if book is found it returns book, else it returns null.
    - searchBySerialNumber() //takes in a serial number, if serial number is null NPE is thrown, if the book with serial number is found that book is returned, if not null is returned.
    - showAllBooks() //returns a StringBuilder object with a list of all books in the library, if no books exists in the library, an empty object is returned.
    - allBooksByYear() //takes in a published year and returns a StringBuilder object with a list of all books from that year, if no books exists with that year, an empty object is returned.
    - isAlreadyInLibrary() //helper method that takes in a book, returns true if that book is in the library and false if it's not.
    - validate() //takes in a book, if the serial number, book title, or author name fields are null throws NPE, else returns true.
- LibraryTests.java
  - Fields:
    - LibraryActions libActions
    - Arraylist<Book> books
    - Book book1, book2, book3, book4, book5, book6, book7, book8
  - Methods:
    - inputBooks //@Before test method used to create eight book objects and instantiate an ArrayList of Book objects.

References

[1] R. Just, and F. Schweiggert: Higher Accuracy and Lower Run Time: *Efficient Mutation Analysis Using Non-Redundant Mutation Operators*. Software Testing, Verification and Reliability, vol. 25, no. 5-7, 2014; 490–507.

[2] R. Just, F. Schweiggert, G. M. Kapfhammer. *MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler*. Proceedings of the International Conference on Automated Software Engineering (ASE), 2011; 612–615.

[3] R. Shweta, B. Suri, S. K. Khatri*: Experimental Comparison of Automated Mutation Testing Tools for Java*. 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 2015.

[4] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin. *Using mutation analysis for assessing and comparing testing coverage criteria*. IEEE Transactions on Software Engineering (TSE) 2006; 32(8):608–624.s

[5] J. H. Andrews, L. C. Briand, Y. Labiche. *Is mutation an appropriate tool for testing experiments?* Proceedings of the International Conference on Software Engineering (ICSE), 2005; 402–411.

[6] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, G. Fraser. *Are mutants a valid substitute for real faults in software testing?* Proceedings of the Symposium on the Foundations of Software Engineering (FSE), 2014; 654–665.

[7] Y. Jia, M. Harman. *An analysis and survey of the development of mutation testing*. IEEE Transactions on Software Engineering (TSE) 2011; 37(5):649–678.

[8] R. Just, F. Schweiggert. *Automating unit and integration testing with partial oracles*. Software Quality Journal (SQJ) 2011; 19(4):753–769.

[9] M.E. Delamaro, J. Maidonado, A.P. Mathur. *Interface mutation: An approach for integration testing*. IEEE Transactions on Software Engineering (TSE) 2001; 27(3):228–247.

[10] H. Coles. *Overview: Mutation Operators*. pitest.org/quickstart/mutators/.

[11] P. Ammann, Paul, and J. Offutt. Introduction to Software Testing: Chapter 1. Cambridge University Press, 2017.

[12] E. W. Wong, and A. P. Mathur. *Reducing the Cost of Mutation Testing: An Empirical Study*. The Journal of Systems & Software, vol. 31, no. 3, 1995, pp. 185–196.

[13] M. Papadakis, and N. Malevris. *An Empirical Evaluation of the First and Second Order Mutation Testing Strategies*. 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, 2010.

[14] A. T. Acree, T. A. Budd, R. A. Demillo, R. J. Lipton, and F. G. Sayward. *Mutation Analysis*. 92. 1979.

[15] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD Thesis, Yale University 1980.

[16] R. A. DeMillo, R. J. Lipton, F. G. Sayward. *Hints on test data selection: Help for the practicing programmer*. IEEE Computer 1978; 11(4):34–41.

[17] P. Ammann, M. E. Delamaro, and J. Offutt. *Establishing Theoretical Minimal Sets of Mutants*. 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, 2014.

[18] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. *Hints on test data selection: Help for the practicing programmer*. IEEE Computer, 11(4):34–41, 1978.

[19] Y. Jia, M. Harman. *An analysis and survey of the development of mutation testing*. IEEE Transactions on Software Engineering (TSE) 2011; 37(5):649–678.

[20] N. K. King, J. Offutt. *A Fortran language system for mutation-based software testing*. Software: Practice and Experience 1991; 21(7):685–718.

[21] Y. S. Ma, J. Offutt, Y. R. Kwon. *MuJava: A mutation system for Java*. Proceedings of the International Conference on Software Engineering (ICSE), 2006; 827–830.
[22] S. Kim, J. A. Clark, J. A. McDermid. *Class mutation: Mutation testing for object-oriented programs.* Proceedings of the Net. Object Days Conference on Object-Oriented Software Systems, 2000; 9–12.
[23] J. Offutt, P. Ammann, and L. Liu. *Mutation Testing Implements Grammar-Based Testing*. Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006), 2006.
[24] J. Offutt, and R. H. Untch. *Mutation 2000: Uniting the Orthogonal*. Mutation Testing for the New Century, 2001, pp. 34–44.
[25] R. Geist, J. Offutt, and F. C. Harris. *Estimation and enhancement of real-time software reliability through mutation analysis*. IEEE Transactions on Computers, vol. 41, no. 5, pp. 550-558, May 1992.
[26] H. Zhu H, P. Hall, J. H. R. May. *Software unit test coverage and adequacy*. ACM Computing Surveys (CSUR) 1997; 29(4):366–427.
[27] P. G. Frankl, S. N. Weiss, C. Hu. *All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness.* Journal of Systems and Software, vol. 38, no. 3, 1997, pp. 235–253.
[28] P. J. Walsh. *A Measure of Test Case Completeness*. Ph.D. Thesis, State University of New York at Binghamton. 1985.
[29] S. Kim, J. A. Clark, and J. A. McDermid. *Investigating the effectiveness of object-oriented testing strategies using the mutation method*. Software Testing, Verification and Reliability, 11(3):207–225, December 2001.
[30] A. J. Ko, T. D. LaToza, M. M. Burnett. *A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants*. Empirical Software Engineering, vol. 20, no. 1, 2013, pp. 110–141.
[31] W. E. Wong and A. P. Mathur. *Fault detection effectiveness of mutation and data flow testing*. Software Quality Journal, 4(1):69–83, March 1995.
[32] J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. *An experimental determination of sufficient mutation operators*. ACM Transactions on Software Engineering Methodology, 5(2):99–118, April 1996.
[33] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian. *Selecting Empirical Methods for Software Engineering Research*. Guide to Advanced Empirical Software Engineering, 2008, pp. 285–311.
[34] Roberts, Donna, and Frederick Roberts. *Variance and Standard Deviation*. Variance and Standard Deviation - MathBitsNotebook(A1 - CCSS Math), mathbitsnotebook.com/Algebra1/StatisticsData/STSD.html.
[35] "Kolmogorov-Smirnov Table." Real Statistics Using Excel, www.real-statistics.com/statistics-tables/kolmogorov-smirnov-table/.