Should Software Testers Use Mutation Analysis to Augment a Test Set?

Ben H. Smith
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
ben_smith@ncsu.edu

Laurie Williams
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
williams@csc.ncsu.edu

Abstract

Mutation testing has historically been used to assess the fault-finding effectiveness of a test suite or other verification technique. Mutation analysis, rather, entails augmenting a test suite to detect all killable mutants. Concerns about the time efficiency of mutation analysis may prohibit its widespread, practical use. *The goal of our research is to assess the effectiveness of the mutation analysis process when used by software testers to augment a test suite to obtain higher statement coverage scores.* We conducted two empirical studies and have shown that mutation analysis can be used by software testers to effectively produce new test cases and to improve statement coverage scores in a feasible amount of time. Additionally, we find that our user study participants view mutation analysis as an effective but relatively expensive technique for writing new test cases. Finally, we have shown that the choice of mutation tool and operator set can play an important role in determining how efficient mutation analysis is for producing new test cases.

1. Introduction

Mutation testing is a testing methodology in which a software tester executes two or more program mutations (mutants for short) against the same test suite to evaluate the ability of the test suite or other verification technique to detect these alterations [1, 2]. A *mutant* (or mutation) is a computer program that has been purposely altered from its original version [3]. Mutants are automatically created via a mutation testing tool using mutation operators. A mutation operator is a set of instructions for making a simple change to the source code [3]. For example, one mutation operator changes one binary operator (e.g. &&) to another (e.g. ||) in an attempt to create a fault variant of the program.

Mutation testing has historically been used to assess the fault-finding effectiveness of a test suite or other verification technique. We use the term *mutation analysis* to denote the process of augmenting an existing test suite to make that test suite *mutation adequate*, meaning that the test suite detects all non-equivalent mutants [4]. After completing mutation analysis, the augmented test suite may reveal latent faults and may detect faults which might be introduced while the system under test is further developed [5].

Mutation analysis is computationally expensive and inefficient [6]. Mutation operators are intended to produce mutants which demonstrate inadequacies in the test set—that is, the need for more test cases [6]. However, some mutation operators produce mutants which cannot be detected by a test suite, and the tester must manually determine these are *stubborn*, or "false positive" mutants. Stubborn mutants make a mutation adequate test suite difficult to

achieve in practice. Additionally, when testers add a new test case, they frequently detect more mutants than was intended, which brings into question the necessity of multiple variations of the same mutated statement.

Using the Goal-Question-Metric (GQM) [7] approach, we formulated our research goal: *The goal of our research is to assess the effectiveness of the mutation analysis process when used by software testers to augment a test suite to obtain higher statement coverage scores.*

Using the goal templates provided by the GQM process, we can rephrase this goal as:

Analyze the **mutation analysis process**
for the purpose of **evaluation**
with respect to **effectiveness**
from the viewpoint of the **software tester**
in the context of **test case augmentation**.

From this reformulation, we elicit four major questions which can help us achieve our goal:

Q1. What is the effect of mutation analysis on coverage scores?
Q2. How long does a developer spend on each new test case while performing mutation analysis?
Q3. Do software testers find mutation analysis useful for augmenting a test set?
Q4. In terms of the set of operators, which operator(s) produces the most new tests?

We conducted two studies to provide insight into these questions. For our first study, we set out to answer Q1-3. We conducted a user study where we observed our participants as they performed mutation analysis for 60 minutes. For this study, our participants used the Jumble[1] mutation testing tool [8]. In our second study, we set out to answer Q4. We performed mutation analysis on the set of mutants created by the MuJava[2] tool, which employs more operators, to empirically determine which operators are the most effective at producing new tests [9].

The remainder of this paper is organized as follows: Section 2 briefly explains mutation testing and summarizes other studies that have been conducted to evaluate its efficacy. Then, Section 3 presents our user study of mutation testing when used by software testers. Next, Section 4 discusses our empirical study on the behavior of mutants of a given operator set. Finally, Section 5 concludes.

2. Background and Related Work

Section 2.1 gives required background information on mutation testing. Section 2.2 analyzes several related works on the technique.

2.1 Mutation Testing

Mutation testing is conducted in two phases. In the first phase, the code is altered into several instances, called mutants, which are then compiled. Mutation generation and compiling can be done automatically, using a mutation engine, or by hand. Each mutant is a copy of the original program with the exception of one atomic change. The atomic change is made based upon a specification embodied in a mutation operator. The use of atomic changes in mutation testing is based on two ideas: the Competent Programmer Hypothesis and the Coupling Effect. The Competent Programmer Hypothesis states that developers are generally likely to create a program that is close to being correct [10]. The Coupling Effect assumes "test cases that distinguish programs with minor differences from each other are so sensitive that they can distinguish programs with more complex differences" [10].

Mutation operators are classified by the language constructs they are created to alter. Traditionally, the scope of operators was limited to statements within the body of a single procedure [11]. Operators of this type are referred to as traditional, or method-level, mutants. For example, one traditional mutation operator changes one binary operator (e.g. &&) to another (e.g. ||) in an attempt to create a fault variant of the program. Recently, class-level operators, or operators that test at the object level, have been developed [11]. Certain class-level operators in the Java programming language, for instance, replace method calls within source code with a similar call to a different

[1] http://sourceforge.net/projects/Jumble
[2] http://cs.gmu.edu/~offutt/mujava/

method. Class-level operators take advantage of the object-oriented features of a given language. They are employed to expand the range of possible mutation to include specifications for a given class and inter-class execution.

In the second phase of mutation testing, a test suite is executed against a mutant and pass/fail results are recorded. If the test results of a mutant are different than the original's, the mutant is said to be *killed* [11], meaning at least one test case was adequate to catch the mutation performed. If the test results of a mutant are the same as the original's, then the mutant is said to *live* or to be *living* [11] indicating that the change represented by the mutant escaped the test cases. *Stubborn*[3] mutants are mutants that cannot be killed due to logical equivalence with the original code or due to language constructs [12]. A mutation score is calculated by dividing the number of killed mutants by the total number of mutants. A mutation score of 100% is considered to indicate that the test suite is adequate [13]. However, the inevitability of stubborn mutants may make a mutation score of 100% unachievable. In practice, mutation analysis entails creating a test set which will kill all mutants that can be killed (i.e., are not stubborn).

In an earlier study, we have shown that mutation analysis is a viable technique to guide test case creation [14]. To leverage mutation analysis for this purpose, the ideal is for every mutant to be detectable and to in fact produce a new test case. The effectiveness of the mutation analysis process, then, can be viewed as the number of new test cases a mutant set produces.

2.2 Related Studies

Offutt, Ma and Kwon contend, "Research in mutation testing can be classified into four types of activities: (1) defining mutation operators, (2) developing mutation systems, (3) inventing ways to reduce the cost of mutation analysis, and (4) experimentation with mutation." [15]. In this sub-section, we summarize the research related to the last item, experimentation with mutation, the body of knowledge to which our research adds.

Several researchers have investigated the efficacy of mutation testing. Andrews et al. [16] chose eight popular C programs to compare hand-seeded faults to those generated by automated mutation engines. The authors found the faults seeded by experienced developers were harder to catch. The authors also found that faults conceived by automated mutant generation were more representative of real world faults, whereas the faults inserted by hand underestimate the efficacy of a test suite by emulating faults that would most likely never happen.

Some researchers have extended the use of mutation testing to include specification analysis. Rather than mutating the source code of a program, specification-based mutation analysis changes the inputs and outputs of a given executable unit. Murnane and Reed [4] illustrate that mutation analysis must be verified for efficacy against more traditional black box techniques which employ this technique, such as boundary value and equivalence class partitioning. The authors completed test suites for a data-vetting and a statistical analysis program using equivalence class and boundary value analysis testing techniques. The resulting test cases for these techniques were then compared to the resulting test cases from mutation analysis to identify redundant tests and to assess the value of any additional tests that may have been generated. The case study revealed that there was only 14-18% equivalence between the test cases revealed by traditional specification analysis techniques and those generated by mutation analysis. This result indicates that performing mutation analysis will reveal many pertinent test cases that traditional specification techniques will not.

Frankl and Weiss [6] compare analysis testing to all-uses testing using a set of common C programs, which contained naturally-occurring faults. All-uses testing entails generating a test suite to cause and expect outcomes from every possible path through the call graph of a given system. The authors concede that for some programs in their sample population, no all-uses test suite exists. The results were mixed. Mutation analysis proved to uncover more of the known faults than did all-uses testing in five of the nine case studies, but not with a strong statistical correlation. The authors also find that in several cases, their tests killed every mutant but did not detect the naturally-occurring fault, indicating that a high mutation score does not always indicate a high detection of faults.

Offutt et al. [13] also compare mutation and all-uses testing (in the form of data flow testing), but perform both on the source code rather than its inputs and outputs. Their chosen test bed was a set of ten small (always less than

---

[3] *Stubborn* mutants are more clearly defined as those living mutants that may or may not be *equivalent* to the original source code. Sometimes, a mutant remains alive and yet cannot be feasibly proven equivalent through formal analysis. [12] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification & Reliability*, vol. 9, pp. 233-262, 1999.

29 lines of code) Fortran programs. The authors chose to perform cross-comparisons of mutation and data-flow scores for their test suites. After completing mutation analysis on their test suites by killing all non-stubborn mutants, the test suites achieved a 99% all-uses testing score. After completing all-uses testing on the same test suites, the test suites achieved an 89% mutation score. The authors do not conjecture at what could be missing in the resultant all-uses tests.

Additionally, to verify the efficacy of each testing technique, Offutt, et al. inserted 60 faults into their source which they view as representing those faults that programmers typically make. Mutation analysis revealed on average 92% of the inserted faults in the ten test programs (revealing 100% of the faults in five cases) whereas all-uses testing revealed only 76% of inserted faults on average (revealing 100% of the faults in only two cases). The range of faults detected for all-uses testing is also significantly wider (with some results as low as 15%) than that of mutation analysis (with the lowest result at 67%).

Ma et al. [17] conducted two case studies to determine whether class-level mutants result in a better test suite. The authors used MuJava to perform mutation testing on BCEL, a popular byte code engineering library, and collected data on the number of mutants produced for both class-level mutation and method-level mutation with operators known to be the most prolific at the latter level. The results revealed that most Java classes will be mutated by at least one class-level operator, indicating that BCEL uses many object-oriented features and that class-level mutation operators are not dependent on each other.

Additionally, Ma et al. completed the mutation analysis process for every traditional mutant generated and ran the resultant test set against the class-level operators [17]. The outcome demonstrated that at least five of the mutation operators (IPC, PNC, OMD, EAM and EMM) resulted in high kill rates (>50%). These high kill rates indicate that these operators may not be useful in the mutation analysis process since their mutants were killed by test sets already written to kill method-level mutants. The study also revealed that two class-level operators (EOA and EOC) resulted in a 0% kill rate, indicating that these operators could be a positive addition to the method-level operators. However, the authors concede that the study was conducted on one sample program, and thus these results may not be representative.

In an earlier study [14], we presented an empirical evaluation of mutation analysis. We performed mutation analysis using the MuClipse[4] mutation testing tool on two open source projects. Our results indicate that 74% of the mutation operators provided by the MuJava framework are effective for the purposes of producing new test cases. The remaining 26% of the operators did not produce new test cases because their mutants were either 1) killed by the original test suites contained in our test beds, 2) stubborn, or 3) were killed while we were trying to kill other mutants.

3. Mutation Analysis User Study

In this section, we use discuss a user study we conducted provide insight into the first three questions posed in Section 1:

Q1. What is the effect of mutation analysis on coverage scores?
Q2. How long does a developer spend on each new test case while performing mutation analysis?
Q3. Do software testers find mutation analysis useful for augmenting a test set?

3.1. Study Procedure

We conducted an observational user study with four software testers, hereby known as "participants." The participants in our empirical study were chosen because they obtained experience with JUnit[5], the Jumble mutation tool (see Section 3.3), and the iTrust[6] web application (see Section 3.4) in a software testing and reliability class conducted at North Carolina State University. Only one participant had any industry experience, but this student's industry experience was not related to mutation analysis. Using TechSmith Camtasia Studio[7] v5.0, we captured

[4] http://sourceforge.net/projects/MuClipse
[5] JUnit is a unit testing framework for Java. http://www.junit.org/
[6] http://sourceforge.net/projects/iTrust
[7] http://www.techsmith.com/camtasia

screen recordings of the participants as they proceeded through the mutation analysis process. The screen recordings helped us gather the metrics which we used to answer Q1-3.

The instructions given to our participants are provided in Appendix B. Before the start of the study, participants were given a brief reminder of the procedure of killing a mutant and of usage of Jumble. Participants were also reminded of the purpose of mutation analysis. We provided an example of a mutant in the iTrust code with a corresponding test with which they could kill the mutant. Participants were also provided a list of the Jumble mutation operators and an example of each type (see Appendix A). After the screen recording was complete, participants were asked to fill out a short survey (see Appendix C) about their feelings on mutation analysis. The results of this survey are discussed in Section 3.7.

Participants were not given a particular "roadmap" for mutation analysis. Each participant was instructed to kill as many mutants as possible in one class at a time, in the order presented in the section "Classes to Mutate" in Appendix B for the duration of 60 minutes. We chose a fixed window of time because we assumed participants would be unlikely to commit to the study if it were unbounded. The choice to set the fixed window at 60 minutes was arbitrary. No participant killed every mutant produced by Jumble. Before moving on to the next Java class, participants were asked to make their best attempt to kill every mutant within a given class on the list in Appendix B. Participants were not asked to record anything about the details of their actions, choices or procedure. There was no restriction given on the number of mutants a participant could kill at a time, nor what the participant was allowed to do to the test code to kill mutants. However, the participants were instructed to refrain from modifying the code under test and to only attempt to kill mutants by adding or modifying tests. In most cases, however, participants followed a procedure that looked something like the following.

1. Execute Jumble against the first (or next) class from the table presented in Appendix B using its corresponding unit tests.
2. Inspect the living mutants as they appear in the console output Jumble provides.
3. If there are living mutants that can be killed, proceed to Step 4. Otherwise, return to Step 1.
4. Write (or edit) a passing test case to kill the mutant(s) in question.
5. Execute Jumble again to ensure that the mutants have been killed.
6. If the mutants have not died, return to Step 4. Otherwise, continue to Step 7. If the only remaining mutants appear to be stubborn, ignore them and continue to Step 7.
7. If there are no more classes on the table in Appendix B, stop. Otherwise, return to Step 1.

3.2. Study Terms

Some mutation tools can be thought of as using a regular expression matcher to modify source code using knowledge and logic pertaining to the constructs of the Java language itself as opposed to operating on compiled binary Java classes[8]. Operating on source code first can lead to two syntactically different expressions within Java being compiled to the same binary object. For example, if a local instance of any subclass of `java.lang.Object` is created, but not initialized within a class, the Java Virtual Machine automatically initializes this reference to `null`. Though developers find automatically initialized variables convenient, the construct causes the code snippets in Figure 1 to be logically equivalent. No test case can discern the difference between a variable which was initialized to null due to the Java compiler and a variable which was explicitly initialized to null by the developer, causing the mutant to be **Stubborn**.

```
//the original code
ITrustUser loggedInUser = null;

//the mutated code
ITrustUser loggedInUser;
```

Figure 1. Logically Equivalent Code

[8] MuClipse operates on source code, Jumble on byte code.

While classifying mutants traditionally contains the categories **killed**, **living** or **stubborn [12]**, we consider it important not only that a mutant dies, but *when* it dies. A mutant which dies on the first execution of the test suite does not yield a new test case, but this might not be true with a different starting test set. Mutants that die with this first execution are called **Dead on Arrival (DOA)**. Additionally, consider a mutant X (created by mutation operator a) that the developer attempts to kill using test case T. Consider that mutant Y (created by mutation operator b) is also killed upon the execution of test case T. Possibly the "two-for-the price-of-one" payoff of test case T may be an anomaly. Or alternatively, perhaps mutation operators a and b generate redundant mutants, or mutants that are often killed by the same test case(s). Mutants killed by a test case written to kill other mutants are called **Crossfire**.

The order in which mutants are inspected will determine which mutants are labeled as crossfire. Although we do not observe a pattern in the order the mutants are presented by the "View Mutants and Results" control in MuClipse, it is possible the orderings we used could introduce some statistical bias. Our intuition is that there exists some number of mutants which would be classified as crossfire regardless of the order in which they were encountered. Crossfire mutants can be thought of as insurance that a given set of programming errors is tested. However, we could possibly reduce the number of mutants that result in the same test case. Usually, a single operator produces several mutants from a single match of a given regular expression within the implementation code. The statistical significance of the order of mutant presentation is outside the scope of this work; future studies can further investigate the effect of crossfire mutants.

When a participant of our user study inspected a given mutant and concluded without killing that mutant, we classify the mutant as **Ignored**. We cannot know which mutant the participant is targeting, so we have to look at the mutation of the next Java class. Since participants were instructed to attempt to kill all mutants for a given class before proceeding to the next, it is a clear indication that the participant has decided to move on from a given mutant (this making that mutant Ignored) when he or she is no longer executing mutation testing on the Java class which contains that mutant. No participant killed every mutant. We classify the mutants that participants could not kill because they ran out of time as **Living**. Finally, **Killed** in the context of this paper means that the mutant was inspected by a mutation tester, and a test was written which detects the mutant.

In summary, we use the following classification scheme for the rest of this paper:

- **Killed.** Mutant which was killed by a test case which was specifically written to kill it.
- **Dead on Arrival (DOA).** Mutant that was killed by the initial test suite found in the test bed.
- **Ignored.** Mutant which a study participant encountered but did not kill.
- **Living.** Mutant which could not be killed due to time constraints (whether or not the mutant was encountered).
- **Crossfire.** Mutant that was killed by a test case intended to kill a different mutant.
- **Stubborn.** Mutant that cannot be killed by a test case due to logical equivalence or language constructs.

Killed mutants provide the most useful information to the software tester: additional, necessary test cases. DOA mutants could have provided us with test cases if our initial test suite had been different, but these mutants might also be considered to be the easiest to kill since they were killed by the initial test suite without any focus by the tester on mutation analysis. Crossfire mutants indicate that our tests are efficient at detecting sets of related errors and may indicate redundant mutants. An operator that has a history of producing a high percentage of Stubborn mutants may be a candidate for not being chosen for mutant generation. The other terms, Ignored and Living, are related to the behavior of our study participants and will be further explained in later sections.

3.3. Jumble

For our user study, we used the Jumble [8] mutation tool. Jumble performs mutation testing on Java byte code. The authors of Jumble chose to emulate the mutation operators used by the mutation tool Mothra [3]. Table 1 presents the Jumble mutation operators and an example of each. Appendix A presents the list of Jumble mutation operators provided to our participants.

Jumble is executed from the command line and produces text-based output which contains a period whenever a mutant has been killed. If the mutant lives, a brief description of the mutation operator which was used and the line the mutation occurred on is printed. Example Jumble output is presented in Appendix B in the midst of the instructions given to the participants. Jumble allows for the mutation testing of multiple Java classes in one execution; however, our participants were instructed to only focus on the mutants of one class at a time. We enforced this restriction on the users because without a clear endpoint for each Java class, we would have no way of

knowing which mutants a participant chooses to ignore. Jumble output indicates the mutants which are living or killed with every execution. In performing mutation analysis, participants ran Jumble many times for a given class to check to see whether the mutant(s) targeted had been killed or not.

Table 1. The Jumble Mutation Operators

Operator	**Examples**	
	Source	**Mutant**
Conditional	`if (x > y)`	`if ( !(x > y))`
Binary Arithmetic	`c = a + b;`	`c = a - b;`
Increments	`i++;`	`i--;`
Inline Constants	`long x = 0l;`	`long x = 1l;`
Class Pool Constants	`public String welcomeStr = "Welcome!";`	`public String welcomeStr = "__jumble__";`
Return Values	`return returnStr;`	`return null;`
Switch Statements	`case 0: i++; case 1: i = 4;`	`case 1: i++; case 0: i = 4;`

3.4. iTrust Application (v4)

Our four study participants performed mutation analysis on the iTrust web-based healthcare application that was created by North Carolina State University graduate students in a Software Testing and Reliability course during 2004-2007. The motivation for iTrust was to provide an example project for use in learning the various types of testing and security measures currently available. Our participants performed mutation analysis on iTrust v4 which was produced in Fall 2007. The students who wrote iTrust v4 were instructed to have a minimum of 80% JUnit statement coverage for each class. We used their JUnit tests as the initial test suite for our empirical study.

For our mutation user study, we randomly chose a set of seven classes in iTrust v4 that had directly corresponding prewritten unit tests. The complexity statistics and starting mutant counts for these classes are presented in Table 2.

All participants started with the same source and test code. iTrust v4 was written to conform to Java 1.6 using a Java Server Pages (JSP) front-end. Each participant conducted mutation analysis using Eclipse[9] v3.3 on an IBM Lenovo laptop with a 1.69 Ghz processor and 1.5 GB of RAM running Microsoft Windows XP. Eclipse and Jumble were executed using Java 1.5 since the test bed source code was written to conform to these standards. iTrust was written to comply with a SQL back-end and thus was configured to interface with a remotely executing instance of MySQL 5.0.

Table 2. Complexity Metrics for iTrust v4 (143 Classes and 7707 LoC for whole project)

Package	**Class**	**LoC**	**Methods**	**Fields**	**Starting Mutants**	
					Living	**DOA**
edu.ncsu.csc.itrust.action	LoginFailuresAction	31	3	3	4	8
	GetVisitRemindersAction	66	3	4	9	6
edu.ncsu.csc.itrust.server	SessionTimeoutListener	30	4	1	3	4
edu.ncsu.csc.itrust.risk.factors	ChildhoodInfectionFactor	24	3	3	3	3
	SmokingFactor	22	3	2	2	3
edu.ncsu.csc.itrust.risk	Type1DiabetesRisks	32	4	0	4	12
edu.ncsu.csc.itrust.validate	BeanValidator	52	7	0	30	7
Total	7 Classes	257	27	13	55	43

[9] http://www.eclipse.org

3.5. Test Effectiveness

Q1. What is the effect of mutation analysis on coverage scores?
Metrics: statement coverage scores before and after

Running Jumble on the classes in our mutation user study produced 98 mutants in total, 43 of which were DOA, and 55 of which were living but detectable. To establish a baseline, the first author killed the same 55 mutants the participants encountered, but without time restriction, and recorded the number of test cases and the statement coverage scores as reported by djUnit[10]. djUnit is an Eclipse plugin which calculates the line and branch coverage for a set of JUnit test cases on a set of Java classes. djUnit also marks the lines which were not executed by the test suite in a view provided by Eclipse. Finally, we observed these metrics for the test suite each participant produced by the end of their participation in our study. We present the statement coverage results in Table 3 with classes in the order participants were given; the baseline column is statement coverage scores when all 55 mutants were killed by the first author.

Most of the classes chosen for our study had high statement coverage scores at the outset. However, as an answer to Q1, our participants were able to improve the average statement coverage score 2-9% (even with the relatively high starting points) by conducting mutation analysis for 60 minutes. In some instances, such as `LoginFailuresAction`, the statement coverage score was improved by most participants by as much as 43%. Participant D was not as familiar with mutation testing as the other participants and therefore ignored many mutants; the coverage scores for participant D's iTrust tests were much lower as a result. Overall, the coverage score improvements are encouraging, since an application with higher statement coverage typically contains fewer faults in its code [18]. We also note here that Jumble did not produce any Stubborn mutants for any class we mutated for our user study.

3.6. Inspection Time

Q2. How long does a developer spend on each new test case while performing mutation analysis?
Metrics: the average inspection time for each classification of mutant, killed mutants per user, new tests per user

[10] http://works.dgic.co.jp/djunit/

We present the number of tests produced by each participant in Table 4. The "Tests" column lists the number of test cases the participant had in their project after performing mutation analysis for 60 minutes. The "New Tests" column is the number of tests each participant had at the end of the study minus the number of tests which were written by the students who wrote iTrust v4 (listed in the "Start" row). As above, the "baseline" row lists the number of tests in the copy of iTrust v4 whose mutants we killed with no time restriction after the study had concluded. Table 4 demonstrates that with 55 mutants, participants were able to create a range of five to 22 new tests, with an average of 10 new tests, for the iTrust v4 classes on which they performed mutation analysis. We found that when the first author killed the same 55 mutants, the process required 14 new test cases and approximately 90 minutes. The first author also followed the mutation analysis approach described in Section 3.1.

Table 4. Number of Test Cases by Participant

Participant	Tests	New Tests
A	462	22
B	446	6
C	447	7
D	445	5
Average	**450**	**10**
Start	440	0
Baseline	**454**	**14**

Table 3. Ending Statement coverage Scores by Participant for Each Class and Baseline

			Participants' Ending Score, 60 Minute Restriction			
Name	**Start**	**Baseline**	**A**	**B**	**C**	**D**
LoginFailuresAction	57%	100%	100%	100%	100%	57%
GetVisitRemindersAction	95%	95%	95%	95%	90%	95%
SessionTimeoutListener	94%	100%	100%	94%	94%	94%
ChildhoodInfectionFactor	100%	100%	100%	100%	100%	100%
SmokingFactor	100%	100%	100%	100%	100%	100%
Type1DiabetesRisks	88%	100%	100%	88%	100%	100%
BeanValidator	96%	100%	96%	96%	96%	96%
Average	**90%**	**99%**	**99%**	**97%**	**97%**	**92%**

When Jumble executes, it produces text-based output which displays Living and Killed mutants. Since the participants were screen recorded for our study, we could measure the amount of time a given participant spent on a given mutant. The average inspection time for each classification of mutant by participant, the number of mutants in each classification by participant, and the number of Jumble executions each user performed are presented in Table 5. We chose to count the first appearance of a mutant in the command line output as the starting time for when the participant has begun to focus on that mutant. Our choice for this measurement is based on the fact that we cannot know which mutant the participant is targeting, nor can we know that the participant is only targeting a single mutant as they proceed with mutation analysis.

Our measurements for the inspection time of a mutant are taken as a difference from one execution of Jumble to another. We can determine the difference in these times by determining the difference in the time codes from our screen recording data. We will now define and explain Equations 1-3, which we used to determine the average mutant inspection time. For Killed mutants, we choose the calculation of inspection time as the execution of Jumble which first displayed that mutant as living until the first execution of Jumble which displays that mutant as Killed as presented in Equation 1.

The calculation of inspection time for Ignored mutants is somewhat less straightforward. We cannot know which mutant the participant is targeting, so we have to look at the execution of Jumble for the next class on the list presented in Table 3. Since participants were instructed to attempt to kill all mutants for a given class before proceeding to the next, it is a clear indication that the participant has decided to move on from a given mutant when he or she is no longer executing Jumble on the Java class which contains that mutant. For Ignored mutants, we choose the calculation of inspection time as the time the mutant is first displayed until the time the mutants for the next class in Table 5 are displayed, as presented in Equation 2.

Even though they were instructed to kill all mutants, every participant ended their participation in the study with some number of Living mutants which they had inspected but could not kill because they ran out of time. In other

words, no participant killed all 55 living mutants. We want to look at how much time was spent on these mutants as well because they could have eventually become either Killed or Ignored by the participant if time permitted. The Living mutants the participant never encountered cannot be included in the calculation of inspection time because the participant never spent any time on them. In other words, a Living mutant would become Ignored if the participant executes Jumble on the next class in the list. We calculate the inspection time of a Living mutant a participant inspected as the time the mutant was first displayed until the end of the study, as presented in Equation 3.

We observed and recorded the inspection time in seconds for each mutant as inspected by each user. For a given participant, the average inspection time for mutants of a given classification was calculated by taking the sum of all the inspection times for mutants of that given classification and dividing by the number of mutants of that classification.

To answer Q2 we find in our user study participants spend about 43 more seconds on average on mutants they kill than on mutants they ignore. We find that the average amount of time spent per mutant overall is around five minutes. We find that participants inspected 26 mutants, on average; in the 60 minute time period they were allotted to conduct mutation analysis. Additionally, we find that our participants produced an average of 10 new test cases while performing mutation analysis. In other words, conducting mutation analysis caused our participants to write one new test case on average every six minutes. Finally, we observe that participants executed Jumble twice per test case produced. This second or third execution was because sometimes a mutation tester can miss the targeted mutant, meaning the tester thought the test case would kill the mutant but the test case does not. Additionally,

$$\text{Inspection Time (Killed)} = \text{Time (First Displayed as Killed)} - \text{Time (First Displayed)} \quad (1)$$

$$\text{Inspection Time (Ignored)} = \text{Time (Next Class First Displayed)} - \text{Time (First Displayed)} \quad (2)$$

$$\text{Inspection Time (Living)} = \text{Time (End of Study)} - \text{Time (First Displayed)} \quad (3)$$

sometimes a mutation tester needs to run Jumble to "take stock" of the remaining living mutants. Due to the experimental setup and the classes chosen, Jumble executions typically require 10-25 seconds.

3.7. Participant Survey

Q3. Do software testers find mutation analysis useful for augmenting a test set?
Metric: common themes in participant answers

We asked our participants to complete a short survey (provided in Appendix C) after they had conducted mutation analysis on the classes in our study. When asked if mutation testing is an efficient way to improve one's test set, all participants said "yes." Two participants stipulated that they were not sure how useful the mutant-killing tests were at actually finding faults.

When asked what the benefits of mutation analysis are, three participants indicated that it causes the tester to quickly create tests the tester might not have otherwise thought of. Two participants also pointed out that there were instances of code which caused an error message to be printed to the console, which reduces testability and that mutation analysis alerted them to these instances.

When asked about the drawbacks of mutation analysis, all participants indicated that mutation analysis consumes a large amount of time. One participant indicated that mutation analysis would be more efficient when

Table 5. Mutant lifetime by participant (seconds)

	Killed		Ignored		Living		
Participant	**Time**	**Quantity**	**Time**	**Quantity**	**Time**	**Quantity**	**Jumble Runs**
A	287	46	197	2	327	7	22
B	352	10	354	3	644	42	15
C	270	16	280	7	349	32	23
D	344	13	237	8	274	34	21
Average	**313**	**21**	**267**	**5**	**399**	**29**	**20**

conducted around the same time unit tests are written in a test-first methodology [19]. Two participants indicated that to kill certain mutants, the tester must create test cases which are incorrect with respect to the specification of the class under test. One participant complained about the time wasted inspecting stubborn mutants in his or her previous usage of mutation analysis.

3.8. Limitations

In this section, we discuss the limitations of the study.

3.8.1. Participants and Code

Our participant study was conducted on a small number of participants who all had somewhat similar backgrounds in mutation analysis and industry. More participants of varying backgrounds should be included in future studies to see variations in the effectiveness of mutation analysis and the average time spent inspecting a mutant. Additionally, a larger, industrial-scale code base should be used to investigate whether the observations we found in our study would be repeated with a larger code base. With regard to the survey, participants are not necessarily experts in mutation analysis and therefore may not be an adequate reflection of how effective the technique is. Additionally, the questions asked of each participant could have been more specific to help gather more information about their opinions on mutation analysis. In conducting this second study, we chose to use Jumble and a later version of iTrust. We changed both tool and system under test to make the preparation time for participants as negligible as possible.

3.8.2. Experimental Setup

A longer testing window could be used to see whether participants hold the same level of productivity if they perform mutation analysis for lengthy timeframes. Future work should also contain observations of users performing mutation analysis at different points in the software process; for instance, comparison of a software system developed by using mutation analysis in conjunction with the test-first model with a software system developed by using only the test-first model. Collecting statement coverage statistics for a software system which does not contain a large and thorough starting test set will provide a much clearer picture as to the improvements to be gained by using mutation analysis in the earlier stages of software development. Future work could also include a mutant set which contains stubborn mutants, because our study contributes no information regarding how long participants spend identifying and inspecting stubborn mutants.

3.8.3. Analysis

Because we cannot discern the participants' intent from the screen recording, many assumptions and data analysis rules had to be enforced to make sense out of the participants' actions. Repeating this study with a more systematic method for recording participants' intentions could yield more insight about how people think about mutation testing and mutation analysis.

4. Evaluation of MuJava Operators

In this section, we use discuss an empirical study we conducted to provide insight into the last question posed in Section 1:

Q4. In terms of the set of operators, which operator(s) produces the most new tests?

We used two versions of three major classes for the Java backend of the iTrust web healthcare application to conduct an empirical study using the MuClipse (see Section 4.2) mutation testing plug-in for Eclipse.

4.1. Study Procedures

To answer Q4, we executed the following procedure:

1. Streamline the already-written test suite by removing redundant or meaningless test cases.
2. Execute the test cases against all generated mutants (a process provided by MuClipse). Record the classification (e.g. live or killed) of each mutant and record the mutation score. Record all mutants killed by the initial test suite as **DOA**. The full classification scheme is discussed in Section 3.2.
3. Inspect the next (or first) living mutant as it appears in the "View Mutants and Results" control in MuClipse. Attempt to write a test case which will kill this (and only this) mutant.
4. If this mutant can be killed, proceed to Step 5. If this mutant cannot be killed due to language constructs or the need to change the source code, record it as **Stubborn** and return to Step 3.
5. Execute the test cases against the remaining living mutants. Record the mutant in question as **Killed**. Record other mutants that are killed by this test case change as **Crossfire**.
6. If there are no more living, killable mutants, stop. Otherwise, proceed to step 3 and repeat.

iTrust utilizes a database management system to store its data. Some mutants would cause insertion of faulty data directly into the database while not affecting the return value from a given function or causing an Exception. Mutants of this type had to be checked using database queries from within the test itself.

4.2. MuClipse

Offutt, Ma and Kwon have produced a Java-based mutation tool, MuJava [9], which conducts both automated subtasks of the mutation testing process in Java 1.4: generating mutants and running tests against created mutants. We created MuClipse by adapting the MuJava [9] testing tool to an Eclipse plug-in, and we have previously conducted an empirical study to observe the behavior of its mutants (see [20]). The differences between MuClipse and Jumble are highlighted in Table 6. The mapping of MuClipse to Jumble operators is presented in Table 7. Additional explanation and an example of each Jumble operator is provided in Appendix A. Because of its more extensive operator set, the fact that MuClipse mutates Java source code, and the system under test in each instance, MuClipse produced many stubborn mutants while Jumble produced none.

In generating mutants, MuJava provides both method- and class-level mutation operators. Additionally, developers can choose which operators will be used to generate mutants and where mutant source files would be placed. MuJava requires unit tests in a Java 1.4 class containing public methods which include the word "test" in their signature. MuJava stores and displays the return values from these test methods (any Java primitive) in the console window during original result collection and mutant result collection. The mutation score and live and killed mutant statistics are displayed at the end of the mutation testing process.

The MuJava application provided the base for the development of MuClipse, an Eclipse plug-in written by the first author. Eclipse Runtime Configurations, which run in Java 1.4, are provided for both the generation of and the testing of mutants. In generating mutants, MuClipse allows the developer to decide which mutant operators to employ and which classes should be mutated (see Figure 2). When testing mutants, MuClipse allows the developer to decide which class's mutants are to be tested; which test suite is to be run against said mutants; and which type of mutant (class- or method-level) operators should be run. MuClipse allows developers the choice of using JUnit[11] test cases as well as MuJava test cases when attempting to kill resultant mutants. JUnit test cases inherit functionality from an abstract TestCase object to provide a result of pass, fail or error. MuClipse stores these results as a boolean true, false or Java Exception when collecting original results or mutant results. JUnit is also available in the form of an Eclipse plug in.

A major component of the mutation analysis process is the management of mutants and their statuses. MuClipse implements an integrated Eclipse View (see Figure 3) which displays mutants and their statuses by Java class and mutant type. This View also contains the overall statistics for live and killed mutants, and the calculated mutation score.

Table 6. Comparison of Jumble and MuClipse

	Jumble	**MuClipse**
Mutates	byte code	source code
Method-level operators	6	15

[11] http://www.junit.org/index.htm

Class-level operators	1	28
Based on	Mothra	MuJava
User interface	Command line	Eclipse plug-in
Written in	Java	Java
Highest compatible Java version	v1.6	V1.4
Design	Industry; optimized for performance	Research; experimental

Some design decisions were made in the MuJava-to-MuClipse modification to increase the efficiency of MuClipse test execution. Firstly, so that developers are not required to track this information manually, MuClipse stores live and killed mutant names for a given execution in a file. Secondly, once a mutant has been marked killed, MuClipse does not execute the test set on it again. Finally, MuJava executed the original source code for each execution of a mutant set to gather the test results for comparison. MuClipse only gathers the test results for the original code once and stores them for comparison during the rest of the mutants of the class under test.

Table 7. Jumble to MuClipse Operator Mapping

Jumble	**MuClipse**
Conditional	COI
Binary Arithmetic	AORB
Increments	AOIS
Inline Constants	n/a
Class Pool Constants	n/a
Return Values	n/a
Switch Statements	n/a

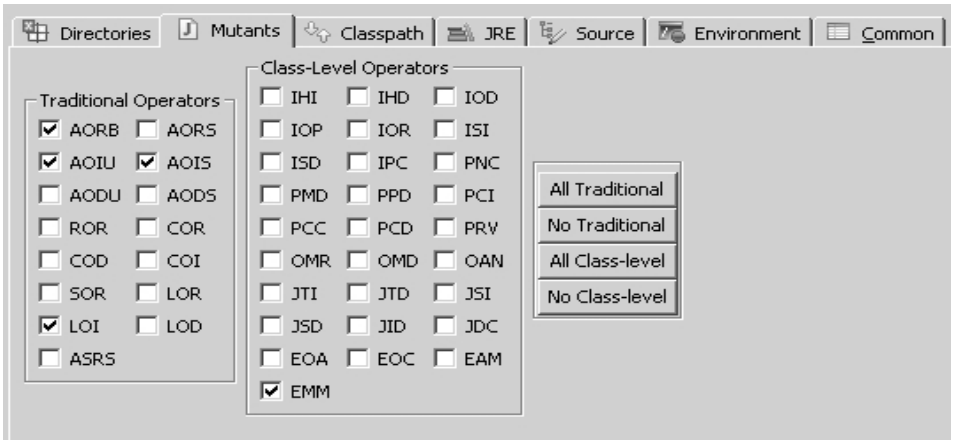


Figure 2. Selecting Operators in MuClipse

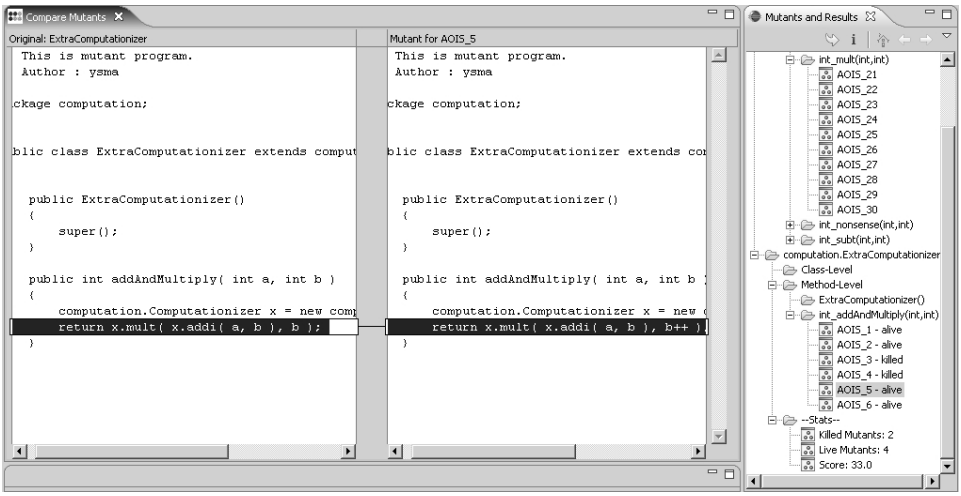


Figure 3. Comparing Mutants to Originals in MuClipse

4.3. iTrust Application (v2)

In answering Q4, we chose MuClipse to expand our possible operator set and to analyze the mutation analysis process with an additional tool. However, due to its development, MuClipse is incompatible with generics, which are a version-specific feature of Java v1.5. As a result, we have chosen iTrust v2, which is written in Java v1.4, and therefore is compatible with MuClipse's mutation engine. iTrust v2 was enhanced by seven two-person teams in the Fall 2006 course. All teams were given the same requirements. In the interest of time, we randomly chose two of the seven teams (which we will call Team A and Team B) and conducted mutation analysis on three classes of their iTrust framework. We chose the classes of iTrust which performed the computation for the framework and dealt directly with the database management back-end using SQL (Auth, Demographics and Transactions). The other classes in the framework act primarily as data containers and do not perform any computation or logic. A comparison of iTrust v2 and iTrust v4 is presented in Table 8. Complexity measurements[12] for each Java class in iTrust v2 for Teams A and B are in Table 9.

Students were instructed to have a minimum of 80% JUnit statement coverage for each class. We used their JUnit tests as the initial test suite for our empirical study. However, students did not always meet this requirement. We found many instances where a student had written a test case which called many methods of the system under test, but contained no oracle. The JUnit framework does not require developers to use an oracle, but djUnit will still count the called methods as executed, even though they have not been tested. The students who developed iTrust were calling methods so they could achieve 80% coverage, as specified in the assignment, without doing the harder work of creating an oracle. As a result, statement coverage dropped from 80% to being within a range of 25%-70% when the first step of the procedure (outlined in Section 4.1) was followed. iTrust was written in Java 1.4 using a Java Server Pages (JSP)[13] front-end. Because the model for the application was to include mostly user interface in the JSP, the three primary classes of the iTrust framework support the logic and processing, including database interaction for the iTrust application. Testing and generation of mutants was executed using Eclipse v3.1 on an IBM Lenovo laptop with a 1.69 Ghz processor and 1.5 GB of RAM running Microsoft Windows XP. Eclipse and MuClipse were executed using Java 1.5 since the test bed source code was written to conform to these standards. iTrust was written to comply with a SQL back-end and thus was configured to interface with a locally executing instance of MySQL 5.0[14].

[12] LoC is every line within the Java implementation code file which is not 1) blank or 2) a comment. LoC calculated using NLOC: http://nloc.sourceforge.net/. Methods refers to public, private, protected, static methods and constructors. Fields refers to public, private, protected and static local variables (declared within the class definition).

[13] http://java.sun.com/products/jsp/

[14] http://www.mysql.com/

Table 8. Comparison of iTrust versions used in our studies

	v2	v4
Release date	Fall 2006	Fall 2007
Versions used in this paper	2	1
Number of Production Classes	~19	143
Number of Test Classes	~7	143
LoC	~2400	7707

Table 9. Complexity Metrics for iTrust v2, Teams A and B

		Line Count for Team:			
Pkg	**Class**	**A**		**B**	
edu.ncsu.csc.itrust	Auth	280		299	
	Demographics	628		544	
	Transactions	123		120	
Total Tested		1031		963	
Total (for all classes, tested and not tested)		2295		2636	
Number of Implementation Classes		20		18	
Number of JUnit Classes		8		5	
		No. fields | No. methods			
Pkg	**Class**	**A**		**B**	
edu.ncsu.csc.itrust	Auth	2	16	2	16
	Demographics	27	22	25	18
	Transactions	2	7	2	7

4.4. Study Results

Q4. In terms of the set of operators, which operator(s) produces the most new tests?
Metrics: the classification of every mutant produced by each operator, the number and type of lines left uncovered by the resultant test set.

4.4.1. Classification Statistics

Though some iterations of the mutation analysis process do not kill any mutants, each iteration classifies a number of mutants. After mutation analysis is complete for a given Java class, totals for each operator were calculated for the number of mutants that were Crossfire, DOA, Killed, and Stubborn. Descriptions of mutation operators used within the MuJava framework can be found in Table 10 and the results of our classification can be found in Table 11.

One answer to Q4 is the following. Of the 1,330 mutants created in total, almost half (560) were spawned by the operator EAM, with ROR (185) and AOIS (104) coming in second and third respectively. By looking at percentages of mutants created, we can see that the COR, AOIU, COI and COD operators produced the highest number of Killed, or useful mutants. We also find that AOIS, JSI, JID and PCD produced the highest number of Stubborn mutants. The traditional operators AODU, AODS, SOR, LOR, LOD and ASRS did not produce any mutants for the iTrust back-end because they mutate language operators that the chosen classes did not contain (e.g., shift, unary logic, unary arithmetic). Similarly, the class-level operators AMC, IHI, IHD, IOD, IOP, IOR, ISI, ISD, IPC, PNC, PMD, PPD, PCC, OMR, OMD, OAN, JTI, JTD, EOA and EOC did not produce any mutants for the iTrust back-end. The operators beginning with I all deal with inheritance features of Java, and the operators beginning with O

and P all deal with polymorphism features. The chosen classes were primarily called upon to perform logic checking and interaction with the database, and thus do not employ many of these language features. The EAM operator replaced one method call to a given class with an equivalent method call, producing a significant anomaly. The coupled nature of the iTrust class structure yielded many mutants by this operator. However, the initial test sets created by both Teams A and B provided simple equivalence class partitioning, which killed these mutants on first execution.

As explained further in the limitations section, the resultant operators with the highest number of Killed mutants could be due to the system under test we have chosen. Back-end code for most web applications makes use of conditional operators to implement behavior. Specifications for a class to perform logic checking and database interaction cause mutation operators effecting conditional operations (COR, COI, COD) to be significantly more useful in producing necessary test cases than those operators related to Java classes or arithmetic (JSI, JID, PCD).

To answer Q4, there seems to be some abstract relationship between the language constructs employed in the implementation code and the effectiveness of mutation operators. However, there is no formal or systematic method to characterize the language constructs used in the implementation code. Aside from some minor patterns we were able to analyze, the results presented in Table 11 cannot be used to make a general statement about which mutation operator is the most effective at producing new tests.

Table 10. Understanding the MuJava Operators, adapted from [14, 21, 22]

Level	Pattern (* = any letter)	Feature (example)
Method	AO**	Arithmetic (+)
	ROR	Relational (==)
	CO*	Conditional (`!`, `&&`)
	S**	Shift (<<)
	L**	Logical (&)
	ASRS	Assignment Short-cut (+=)
Class	AMD	Encapsulation
	I**	Inheritance
	P**	Polymorphism
	O**	Overloading
	J**	Java-Specific (e.g. `private` keyword)
	E**	Inter-Object

Table 11. Classification by Operator (Teams A and B)

Operator	Description	Killed	Stubborn	DOA	Crossfire	Total
AOIS	Insert short-cut arithmetic ops.	9	62	7	26	**104**
AORB	Replace equivalent arithmetic ops.	2	0	3	7	**12**
COD	Delete unary conditional ops.	6	0	32	2	**40**
COI	Insert unary conditional ops.	17	0	66	10	**93**
COR	Replace equivalent binary ops.	15	6	26	5	**52**
LOI	Insert unary logic ops.	1	7	31	20	**59**
EAM	Change method accessor	20	53	345	142	**560**
JSI	Insert static modifier	3	24	0	24	**51**
JID	Remove variable initialization	0	1	2	0	**3**

PCI	Insert type cast operator	4	0	28	9	**41**
AOIU	Insert basic arithmetic ops.	5	1	16	6	**28**
ROR	Replace relational ops.	10	26	99	50	**185**
PCD	Delete type cast operators	0	35	0	38	**73**
JSD	Delete static modifier	0	0	0	1	**1**
EMM	Change method modifier	0	0	0	2	**2**
PRV	Replace reference with equivalent	0	0	0	25	**25**
JDC	Default constructor creation	0	0	1	0	**1**
Total		**92**	**215**	**557**	**365**	**1330**

4.4.2. Unexecuted Statements

To answer Q4, we also executed djUnit to indicate which lines were not executed by our test suites after the mutation analysis process. That is, after all non-equivalent mutants had been killed for each class, we executed djUnit and examined each line of code that was left not executed. Each line of code dJUnit marked as not executed was classified into one of the following groups.

- **Body.** Lines that were within the root block of the body of a given method and not within any of the other blocks described, except for try blocks (see below).
- **If/Else.** Lines that were within any variation of conditional branches (if, else, else if, and nested combinations).
- **Return.** Java `return` statements (usually within a constructor, getter or setter).
- **Catch.** Lines that were within the catch section of a try/catch block. Since most of the code in iTrust belongs in a try block, only the catch blocks were counted in this category.

Most lines not executed in the iTrust classes under test fell into a catch section of a `try/catch` block (see Table 12), corresponding to the fact that mutation operators for Java Exceptions have yet to be developed (though researchers have discussed that possibility [17]). Since no mutants were created which change, for instance, which Exception is caught, no test needs to be written to hit the catch block and therefore the catch block remains not executed.

A number of lines were additionally found within individual `if` statements and within the body, indicating sections of code which might more likely be executed in a real world setting that were left not executed by the mutation analysis process. The procedure followed precluded fixing errors in the original source code, which caused many of these `if` and `body` instances. For instance, when a statement fell after a statement which threw an exception, we could not write a test to reach the statement after the exception-throwing statement, because we could not modify the source code.

In answering Q4, we find that mutation analysis can miss certain areas of code depending on which operators result in stubborn mutants and what the operator set mutates. The chosen set of operators will determine the areas of the source code which will receive the mutation tester's attention.

Table 12. Not Executed Lines by Team and Class

Team	Class	catch	if/else	return	body
A	Auth	32	9	3	0
	Demographics	28	50	2	69
	Transactions	11	4	0	0
B	Auth	28	9	0	5
	Demographics	39	27	0	4
	Trans	11	5	0	0
Totals		149	104	5	78

4.4.3. Limitations

Our empirical results apply only to the iTrust application and to the MuClipse tool and may not extrapolate to all instances of mutation analysis. iTrust is a relatively large academic code base and larger than the software used in many other mutation testing and mutation analysis studies but still small relative to industrial software. Also, due to the expensive nature of mutation analysis and our iterative experimental setup, we limited our test set to the three classes from only two teams listed in Section 4.2. Testing all of the classes of the iTrust framework from all of the teams would yield more information about mutation analysis. Furthermore, the source code for iTrust was similar among all teams in that it belonged to the back-end of a web-based application—mutation analysis on other systems may yield different numbers of mutants in each of our classification categories. Additionally, the set of mutation operators provided by the MuJava framework is more inclusive by providing object-oriented operators, but cannot be determined as being representative of every possible operator that could be written. Future work could include performing the procedure detailed in our work on additional, larger systems. In addition, the expensive nature of our experimental process precludes us from a more real-world procedure of fixing source code that leads to stubborn mutants, which could lead to more insights about the limitations of mutation analysis. Furthermore, removing redundant tests may have introduced a difference in mutation score; two equivalent test sets may have different mutation scores with the same statement coverage score. Some inconsistencies in the starting mutation scores may have been introduced during this step.

5. Conclusion

Our experiences over the course of conducting these two studies are as follows:

- The apparent worth of a mutation operator can be measured by its behavior across multiple classes and test executions. That is, we find it necessary to track the number of mutants in each of our classification scheme for mutation operators to help gather information about mutation operators' behavior in the future.
- Mutation testing can be thought of as automating one part of a tester's thought process: imagining faults which could be in the code now or in the future. Software developers should incorporate mutation analysis into the unit phase of software, especially when using the test-first paradigm, because "imagining" faults early will help ensure the correctness of the developed component.
- Choosing mutation testing tools which operate on the byte code is prudent, since short execution time ensures an adoption of mutation analysis by software testers.
- Experimental mutation systems are worthwhile because they act as an avenue to introduce mutation concepts such as mutating at the object level. However, future operator sets should focus on covering a vast variety of language constructs, rather than focusing on how these constructs are changed.
- Additionally, larger mutation operator sets consume more mutation time, but performing mutation analysis increases statement coverage scores regardless of the size of the operator set. While there is probably some minimum number and type of operators that are required to achieve a benefit from mutation analysis, we contend that no existing mutation operator set is exhaustive. Therefore, if the technique is to be used by software testers, then mutation speed and efficiency should be chosen over the number or type of mutation operators.

We conducted two empirical studies on mutation analysis. Our first study shows that though our participants expressed some concerns about the efficiency of mutation analysis, they were able to effectively use mutation analysis for a relatively short amount of time to produce new test cases and improve statement coverage scores. We note here that the expensive nature of mutation could be mitigated by an improved set of tools for automated test data generation; however, we also contend that mutation analysis forces testers to manually examine the source code (to find the location and nature of the mutant) in addition to augmenting test cases. This manual source examination could prove to be more useful than automated test generation for a validation and verification team. Our second study demonstrates that tool and operator choices, as well as the system under test, can have a strong impact on the ability of mutation analysis to create new tests. We find that mutation tools and mutation operators which mutate the source code end up producing more stubborn mutants, though this could be attributable to the system under test. Finally, we have found that even an extensive research-based mutation operator set can still miss certain mutations which could point the mutation tester to other areas of code which should be tested.

While most other studies conducted on mutation analysis to date are based on a research team conducting mutation analysis (see Section 2), the goal of our research was to assess the effectiveness of the mutation analysis process when used by *software testers* to augment a test suite to obtain higher statement coverage scores. The idea behind this experimental setup is to see what the variations are in the way mutation analysis is used when not in the

hands of an expert or a researcher in the mutation analysis field. We find that our participants view mutation analysis as an effective but expensive technique for augmenting an existing test suite. Finally, we have shown that the choice of mutation tools and mutation operators plays an important role in determining how effective mutation analysis is for producing new tests. Given that mutants are representative of real faults, we find that mutation analysis can be used as an effective way to improve the fault-finding ability of a test set.

6. Acknowledgements

Funding for this research was provided by an IBM Faculty Award. Additionally, we would like to thank the North Carolina State University RealSearch group for their helpful comments on the paper. This work is supported in part by the National Science Foundation under CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. References

[1] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[2] M. Trakhtenbrot, "New Mutations for Evaluation of Specification and Implementation Levels of Adequacy in Testing of Statecharts Models," *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 151-160, 2007.

[3] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King, "An extended overview of the Mothra software testing environment," in *The Second Workshop on Software Testing Verification, and Analysis*, Banff, Alta. Canada, 1988.

[4] T. Murnane, K. Reed, T. Assoc, and V. Carlton, "On the effectiveness of mutation analysis as a black box testing technique," in *Software Engineering Conference*, Canberra, ACT, Australia, 2001, pp. 12-20.

[5] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *27th International Conference on Software Engineering*, St. Louis, MO, USA, 2005, pp. 402-411.

[6] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: An experimental comparison of effectiveness," *The Journal of Systems & Software*, vol. 38, pp. 235-253, 1997.

[7] V. R. Basili, "Software modeling and measurement: the Goal/Question/Metric paradigm," 1992.

[8] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, M. Utting, and R. T. Ltd, "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests," *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 169-175, 2007.

[9] J. Offutt, Y. S. Ma, and Y. R. Kwon, "An experimental mutation system for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 1-4, 2004.

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34-41, 1978.

[11] R. T. Alexander, J. M. Bieman, S. Ghosh, and J. Bixia, "Mutation of Java objects," in *13th International Symposium on Software Reliability Engineering*, Fort Collins, CO USA, 2002, pp. 341-351.

[12] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification & Reliability*, vol. 9, pp. 233-262, 1999.

[13] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Practice and Experience*, vol. 26, pp. 165-176, 1996.

[14] B. H. Smith and L. Williams, "On Guiding the Augmentation of an Automated Test Suite via Mutation Analysis," NCSU CSC TR-2008-9, accepted to Empirical Software Engineering.2008.

[15] J. Offutt, Y. S. Ma, and Y. R. Kwon, "The class-level mutants of MuJava," in *Proceedings of the 2006 International Workshop on Automation of Software Tesing*, Shanghai, China 2006, pp. 78-84.

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *Proceedings of the 27th international conference on Software engineering*, St. Louis, MO, USA, 2005, pp. 402-411.

[17] Y. S. Ma, M. J. Harrold, and Y. R. Kwon, "Evaluation of mutation testing for object-oriented programs," in *28th International Conference on Software Engineering*, Shanghai, China, 2006, pp. 869-872.

[18] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys*, vol. 29, 1997.

[19] K. Beck, *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.

[20] B. H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pp. 193-202, 2007.
[21] Y. S. Ma and J. Offutt, "Description of class mutation mutation operators for Java," 2006.
[22] Y. S. Ma and J. Offutt, "Description of method-level mutation operators for Java," 2005.

Appendix A. Jumble Mutation Operators and Explanation

Jumble Output	Explanation	Example Original	Mutation
`CP[36] ":" -> "___jumble___"`	The String literal for the colon symbol was replaced with the String literal ___jumble___. The CP[X] indicates that the mutation occurred on the String literal on the Xth character on that line.	`String x = “”;`	`String x = “___jumble___”;`
`changed return value (areturn)`	The return statement on this line was altered to return null.	`return someObject;`	`return null;`
`negated conditional`	The conditional expression on this line has been mutated to be its logical complement.	`if (a > b)`	`if (!(a > b))`
`CP[71] 251.0 -> 503.0`	The numerical constant on the left was replaced with the numerical constant on the right.	`private static double aConstant = 251.0;`	`private static double aConstant = 503.0;`
`9 (#) -> 10 ($)`	The inline constant on this line (on the left) was altered to become the constant on the right.	`private static final int x = 9;`	`private static final int x = 10;`
`0L -> 1L`	The literal of type long was mutated from the value on the left to the value on the right	`if (mid > 0l)`	`if (mid > 1l)`
`++ -> --`	The arithmetic operator on the left was mutated to become the arithmetic operator on the right.	`for (int i=0;i<3;i++)`	`for(int i=0;i<3;i--);`

Appendix B.

Mutation Testing User Study – Instructions

Welcome

Welcome to the Mutation User Study. We very much appreciate your help and thank you for sacrificing your time to participate and get your $35 gift certificate. As you proceed through your handbook, be aware that Ben is available to answer your questions at any time (even during recording!).

To begin, we will share an overview of what we hope to accomplish today in order.

1. Reminder of mutation concepts and terms
2. Brief explanation of what we hope to learn from this study
3. Refresher on how to kill a mutant
4. Information on how to complete your part in this study (a comprehensive list)

Mutation Testing

Mutation testing is a method to assess the validity of your test set. The source code is altered into several instances, called **mutants**, which represent a developer fault. When a mutant is detected by the test set, it is said to be **killed**. Otherwise, it is said to be **living**. Some living mutants cannot be killed; these are called **stubborn** mutants. The number of mutants killed divided by the number of mutants remaining is called the **mutation score**. A high mutation score indicates that your test set is very fault-perceptive. Additionally, our research has begun to focus on the concept that *writing* test cases to detect mutants can be used as a way to make your test cases better at finding faults.

What We Hope to Learn

Mutation testing has historically been very inefficient. As computing power increases, it is now becoming feasible to integrate the technique into mainstream test-driven development. Many studies have been conducted which evaluate the benefits and issues related to mutation testing, but no researchers have investigated how people could actually *use* it. From watching you kill mutants, we hope to learn how effective advanced developers could be at using mutation testing to increase their coverage score and the fault-perception of their test suites. At the end, we will ask you some questions about mutation testing and your previous development experience.

How to Kill a Mutant

We will now walk you through killing one mutant. Our hope is that this step-by-step procedure will act as a refresher to the thinking required by mutation testing.

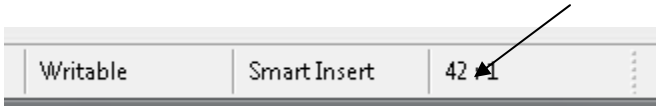
1. Ensure you are in the Eclipse "Java Perspective".
2. Open the files `edu.ncsu.csc.itrust.action.AddOfficeVisitAction` (in `/src`) and `edu.ncsu.csc.itrust.action.AddOfficeVisitActionTest` (in `/unittests`).
3. Right click on `AddOfficeVisitAction` and click on `Jumble -> Jumble Class`. You should get an error informing you that the system cannot find `context.xml`, open the run dialog box and type `iTrust` into the `Project` field. After running Jumble, you should get the following output:

```
Mutating edu.ncsu.csc.itrust.action.AddOfficeVisitAction
Tests: edu.ncsu.csc.itrust.action.AddOfficeVisitActionTest
Mutation points = 4, unit test time limit 8.41s
M FAIL: edu.ncsu.csc.itrust.action.AddOfficeVisitAction:30: CP[75]
"visit id: " -> "___jumble___"
..M FAIL: edu.ncsu.csc.itrust.action.AddOfficeVisitAction:39:
changed return value (areturn)

Score: 50%
```

We will be killing the second mutant, which reads `changed return value (areturn).`

4. To observe this mutant, open AddOfficeVisitAction and double click on the line numbers in the lower-right hand corner of the screen, as can be seen in the following figure. A dialog box will appear asking you to enter the line number. Type 39 and press enter.



The output from the code above means that Jumble changed the line

```
return factory.getAuthDAO().getUserName(pid);
```

To

```
return null;
```

We have included a table at the end of this handbook you can use for interpreting Jumble output.

5. Killing this mutant requires writing a test case which checks the output value from the action method `getUserName()`. It is easier in this case to use a test case which already exists to kill the mutant; we will use `testAddEmpty()`.
6. Open `AddOfficeVisitActionTest`.
7. At the end of `testAddEmpty()`, uncomment the line

```
assertEquals("Random Person", action.getUserName());
```

8. Run Jumble on `AddOfficeVisitAction`. Your output should not contain the description of this mutant because we have just killed it. Additionally, your mutation score should now be 75% instead of 50%, resulting in the following output.

```
Mutating edu.ncsu.csc.itrust.action.AddOfficeVisitAction
Tests: edu.ncsu.csc.itrust.action.AddOfficeVisitActionTest
Mutation points = 4, unit test time limit 8.71s
M FAIL: edu.ncsu.csc.itrust.action.AddOfficeVisitAction:30:
CP[75] "visit id: " -> "___jumble___"
...
Score: 75%
```

Classes to Mutate

In this order. Again, do not move on until you have attempted to kill all the mutants from a given class.

```
edu.ncsu.csc.itrust.action.LoginFailuresAction
edu.ncsu.csc.itrust.action.GetVisitRemindersAction
edu.ncsu.csc.itrust.server.SessionTimeoutListener
edu.ncsu.csc.itrust.risk.factors.ChildhoodInfectionFactor
edu.ncsu.csc.itrust.risk.factors.SmokingFactor
edu.ncsu.csc.itrust.risk.Type1DiabetesRisks
edu.ncsu.csc.itrust.validate.BeanValidator
```

Procedural Information

Please read this entire list before moving on to the next section. Ask questions if you do not understand or want something clarified.

- You will be completing the above process, or something similar to it for every mutant in the list of Java classes attached to this handbook. To Jumble the class the first time, you will have to right-click it, go to `Jumble -> Jumble Class`, and type `iTrust` into the `Project` field. From then on, it is saved as a runtime configuration until you Jumble the next class.
- The procedure will begin after you have finished reading this list and tell Ben you would like to start; you are allotted 60 minutes from that point.
- The list of interpretations of Jumble mutations is also attached; we recommend you study it for a few moments before beginning, but you will have access to it throughout the procedure.
- Some mutants you encounter will either be impossible to kill or extremely difficult to kill. We encourage you to try your best to kill each mutant before moving on, but *you* make the final decision about whether a mutant is stubborn. You do not need to indicate that you have decided a given mutant is stubborn.
- Please kill every mutant you can in a given class before moving on to a different class.
- If you wish, you can attempt to write a test case which will kill multiple mutants, but this is not required.
- You can ask Ben for help in understanding the research procedure, but after you begin, no help will be given regarding how to kill a given mutant.
- You will be asked to run your dJUnit coverage tests at periodic intervals and display the coverage report. Please display the coverage for the class you are Jumbling.
- Do not worry about your progress or effectiveness. It is just as interesting to us that you kill a lot of mutants as that you only kill a few.
- The screen recording program may cause screens to appear slowly or slightly differently than you are used to; please do your best to ignore this. Also please do not do anything to the screen recorder at any point during the process.
- You can change the test code in anyway you please to kill a mutant, including adding an assert statement, changing an existing assert statement, or creating a new test case.
- When your time is up, you will be asked to complete a short user survey. This survey will give us an idea of your previous development experience and your perception of mutation testing. Return this form to Ben when you are finished.
- Before Jumbling your first class, please run `DBBuilder` and `TestDataGenerator`.

Appendix C. Mutation Testing User Study — User Survey

1. How familiar are you with mutation testing? With testing in general? Do you have any industry experience? Please be specific.
2. Do you feel that mutation testing is an efficient way to improve your test set? Please explain your answer.
3. In your view, what are the *benefits* of using mutation testing? Please provide examples.
4. In your view, what are the *drawbacks* of using mutation testing? Please provide examples.