Transactions on Engineering and Computing Sciences - Vol. 11, No. 5
Publication Date: October 25, 2023
DOI:10.14738/tecs.115.15424.
Falah, B., Abufardeh, S., & Bourhnane, S. (2023). Comparison of Mutation Testing and Manual Seeded Testing in Regression Test Selection. Transactions on Engineering and Computing Sciences, 11(5). 16-37.
SERVICES FOR SCIENCE AND EDUCATION

Comparison of Mutation Testing and Manual Seeded Testingin Regression Test Selection

Bouchaib Falah
School of Science and Engineering,
Al Akhawayn University, Ifrane, Morocco

Sameer Abufardeh
Computer, Electrical, and Software Engineering Department,
Embry-Riddle Aeronautical University,Prescott, AZ, USA

Safae Bourhnane
School of Science and Engineering,
Al Akhawayn University, Ifrane, Morocco

ABSTRACT

Mutation testing and manual seeding testing are two distinct techniques employed in software testing. Mutation testing involves the introduction of artificial faults, called mutations, into the codebase, followed by executing the existing test suite against these mutated versions. On the other hand, manual seeding testing entails deliberately inserting faults into the source code by an individualor a group to monitor detection and removal rates and estimate the number of remaining faults. This paper introduces both techniques, discusses their advantages and disadvantages, and compares themin terms of test case selection using another testing technique, regression testing, which ensures thatpreviously developed and tested software performs consistently after modifications and whose primary objective is to minimize testing costs by selecting and executing the minimum number of testcases.

Keywords: Mutation testing, error-seeding techniques, regression testing, test case selection, MuJava,mutation operators.

INTRODUCTION

Regression testing is the process of evaluating and revalidating modified software to detect whether new defects have been introduced into the previously tested version of that software and to build confidence in the correct performance of the modified software.

Let P be a program and P′ be a modified version of P. Let T be a set of test cases used for P. The simplest regression testing strategy is to retest all when developers rerun the whole test suite T for P′.Implementing the process is very simple, but this approach may consume excessive time and resources,especially when only a small part of the application P is modified. To reduce some of this expense without minimizing testing effectiveness, researchers suggested that tests that cannot possibly be affected by the modifications can be ignored. Thus, they have proposed

regression test selection. Theyselect only a subset of test cases, T′, from the test suite T and run it for P′. If the subset, T′, is small enough, significant savings in time and budget are achieved.

To improve the effectiveness of regression testing, numerous researchers have proposed various testcase selection methods in recent years. These selection techniques were based on some criticality or perceived effectiveness. Many of these researchers examined different selection test techniques suchas minimization, safe, dataflow, control flow, random, and retest-all, focusing on their abilities to reduce the cost of testing by reducing the test suite and to detect defects.

Mutation testing is a fault-based testing technique used to find the effectiveness of test cases and assure software quality. Mutation testing is an automated technique that involves introducing small artificial faults, known as mutations, into the codebase and then executing the existing test suite againstthese mutated versions. The purpose is to evaluate the effectiveness of the test suite in detecting theintroduced faults. If a test case fails to detect a mutation, it indicates an inadequate test case.

On the other hand, the seeding errors technique is a strategy where faults are intentionally inserted into a source code by one person or a group for the purpose of monitoring the rate of detection and removal and estimating the number of faults remaining in the program. This testing process involves deliberately seeding or adding a certain number of falsely generated errors into the source code of a program, and test cases are run to detect defects.

In our research, we introduced mutation testing techniques and errors seeded technique and their advantages and disadvantages in selecting test cases. Then we compared them by conducting an empirical study, in which we calculated:

- The ratio of killed mutants by the total number of killed and alive mutants for mutation technique.
- The ratio of the number of killed seeded errors to the total number of detected and non-detectederrors for the errors seeding technique.

RELATED WORK

To test any software system, testers have to create test cases. Test cases are created to detect bugs in the system and to verify the correctness of the implementation and the software system. However,the creation of test cases still poses the question of whether those test cases are effective or not andwhether they satisfy the requirements that drove the implementation.

To expose this weakness in creating test cases, mutation testing was originally proposed by Richard Lipton as a student in 1971, resulting in mutation testers appearing on several different platforms. The first mutation testing was developed and published by DeMillo, Lipton, and Sayward in April, 1978 [1].Traditionally, software testing has been an ad/hoc process done by all testers [2]. Testers rely on theirprior knowledge, intuition, and experience to create test cases, and if the program passes those test cases, they then conclude that the program is correct. Certainly, the test cases created based on priorexperience, knowledge, and intuition are not sufficient to significantly augment our confidence in thecorrectness of the functionality of the

program. Accordingly, modern testing techniques were developed to increase the confidence of the testers in the correct functioning of their programs by exercising the program under different circumstances. The first mutation tool was implemented by Tim Budd as part of his Ph. D in 1980 [3].

There has been a renaissance of mutation analysis in computer science. Several researchers used numerous popular mutation tools. There have been many researchers who work on different kinds oftechniques seeking to turn mutation testing into a practical testing approach by creating several mutation testings tools. PIMS [4, 5], an early mutation testing tool, launched the general process that was used in mutation testing of creating mutants of Fortran IV programs by accepting test cases from the users and then executing them on the mutants to detect the number of mutants that were killed.This process was adopted and extended late in 1987 in the Mothra mutation toolset [6, 7], where themutation analysis was used as a basis for building an incorporated software environment. Many othermutation testing tools have been implemented after Mothra, and many techniques have beendeveloped to determine whether test data can discover specific defects (e.g., Mutation analysis [1]). Ma, Offutt, and Kwon [8] built a mutation testing tool based on Mutation Schemata Generation (MSG)and byte code translation methods. The aim of their work is to reduce the execution cost of mutation testing for OO programs by determining how fast these methods are compared to the compilationapproach.

DeMillo and Offutt [9] believe that their work was the first work that attempted to automatically generate test data to kill mutant programs. Their technique was based on mutation analysis. They used the Constraint Based Testing (CBT) method, which allows them to combine path-coverage techniques and symbolic execution with mutation analysis [9, 10]. To create test cases that can kill mutant programs, constraints are automatically generated by a fully implemented set of tools called Godzilla, which is integrated into the Mothra software testing system [7, 11]. To perform mutation analysis, Offutt, Untch, and Harrold [12] implemented a Mutation Schema Generation (MSG) that enables them to encode all mutants into one source-level program rather thaninterpreting different versions of the same program. Their result showed that MSG mutation systems are faster and cheaper to build than interpreter-based systems.

Mutation testing is a widely discussed technique among researchers in the field of software testing. However, there seems to be less emphasis on the seeding errors technique, which is usually performed by individual developers during the development process. Seeding errors involves intentionally introducing faults into the codebase to understand how the system responds to differentkinds of failures and uncover potential weaknesses.

REGRESSION TESTING

To improve the effectiveness of regression testing, numerous researchers have proposed various testcase selection methods in recent years. These selection techniques were based on some criticality or perceived effectiveness. Many of these researchers examined different selection test techniques suchas minimization, safe, dataflow, control flow, random, and retest-all, focusing on their abilities to reduce the cost of testing by reducing the test suite and detecting defects.

Minimization Techniques

The aim of the minimization technique is to select a minimal test suite, T^', that covers all modified oraffected elements of P^'[13, 14].

Graves, Harrold, Porter, Kim, and Rothermel [13] investigated the cost and benefits of several selection testing techniques including minimization techniques. They concluded that this technique uses a small number of test cases, but it is very weak in detecting all defects.

Porter, Kim, and Rothermel [15] investigated how the number of changes between P and P' affects the effectiveness and the performance of the minimization testing technique. To perform their experiment, the researchers implemented a regression selection testing tool that selects a minimal test suite, T', that contains at least one test case for every node in the control flow graph of the originalapplication version, P, that was modified in the new version P'. Their experiment showed that the minimization technique selected 1.5% of the original test cases and achieved 44.2% absolute effectiveness [15]. Therefore, this technique selects a few test cases and is more effective as the number of changes increases.

Coverage Techniques

Coverage techniques, either dataflow or control flow, emphasize the use of structural coverage criteria; they attempt to locate program components, such as statements or definition-use pairs, thathave been modified or may be affected by modifications and select test cases from the test suite T that exercise those components.

Definition-Use pair (DU) consists of a definition and a use for a variable such that at least one DU pathexists from the definition to the use.

Several regression selection testing techniques are based on dataflow analysis and testing techniques.Dataflow test selection techniques identify definition-use pairs that are new in P', modified for P', or deleted from P, and select test cases that exercise those pairs.

Hutchins, Foster, Goradia, and Ostrand [16] tried to obtain meaningful information about the effectiveness of dataflow and control flow techniques for fault detection. In their experiment, the researchers seeded realistic faults in 130 program versions derived from seven moderate-size C baseprograms. For each faulty program, several thousand test suites were generated, and the relationship between fault detection and coverage was examined. Based on their experiments, they concluded that test suites achieving coverage levels over 90% usually showed significantly better fault detectionthan test cases chosen randomly of the same size and that 100% coverage, either dataflow or controlflow, is not an indication that testing has been adequate.

Ad Hoc/Random Techniques

When testers are under time pressure, and no selection tool is available, they often select their test suite, T', randomly based on their skills, knowledge, and experience.

Graves, Harrold, Porter, Kim, and Rothermel [13] investigated the cost and benefits of random techniques. They created a tool that randomly selects n% of the test cases from test suite T. Their experiment showed that the effectiveness of the random techniques increased with test suite size, but the rate of increase diminished as size increased. By comparing the random technique with the minimization technique, the researchers found that, on average, the random technique with a slightlylarge test suite leads to an equivalent fault detection result as the minimization technique.

Kim, Porter, and Rothermel [15] applied their experiment on Siemens programs taken from an experiment done by Hutchins et al. [16] to evaluate the effectiveness of random testing techniques. Their result showed [15] that the median relative effectiveness is always over 90% when the testing time interval is 1. As the testing time interval increases, relative effectiveness also increases. Therefore, this technique has high effectiveness at low testing time intervals and increases effectiveness as testing time intervals increase. In addition, when the testing time interval is high, a small test suite randomly selected is the most cost-effective if the cost of missing faults is not too high. Thus, their analysis showed that random techniques are surprisingly cheap and effective, and effectiveness was not greatly affected by change activity [14].

Safe Techniques

Safe techniques are selection techniques that select, under some conditions, every test case in T thatreveals defects in the modified software. The strength of these techniques is that they omit no fault- revealing tests, and thus, these techniques guarantee that the faults revealed by this subset will be the same as those revealed by running the entire test suite T (using retest-all techniques). Kim and Porter [14] presented an empirical study using historical execution data to compare safe techniques with retesting all. Their experiment used the DejaVu tool implemented by Rothermel andHarrold [17, 18]. Based on investigating the cost and benefits of several regression testing selections under two different software models, their study showed that Safe techniques select a large, varied number of test cases with a total effort less than that required by retest-all. As a result, under certainconditions, safe techniques guarantee that the selected test suite T' detects any defects that test-all would have with less effort.

As an example of safe techniques, Rothermel and Harrold [19] developed a good used control flow graphs for P and P' to select test cases that execute changed code from the original test suite T. Theirtechnique selects tests based on information about the code of the program P and the modified version P'. Their algorithms indicate that, under certain conditions, the test suite T' includes test casesfrom T that can disclose faults in the modified version P', and thus their algorithm is safe for controlled regression testing under those conditions. Furthermore, Rothermel and Harrold [20] empirically investigated using safe regression test selection techniques by performing five studies. These studies illustrated that safe test selection techniques canreduce the cost of regression testing and yield greater savings when applied to larger and more complex programs than when applied to smaller and simpler programs. Also, these techniques were able to produce results in detecting faults equivalent to those achieved by retest-all techniques and thus can be cost-effective. The cost-effectiveness can vary based

on some factors (e.g., the cost of analysis necessary, the cost of executing and validating test cases).

MUTATION TESTING TECHNIQUES

Mutation testing is a fault-based testing technique used to find the effectiveness of test cases and assure software quality. Mutation testing assumes that a program will be well-tested if all simple faultsare detected and removed. This testing technique is done by seeding artificial faults into the implementation (original source code) and checking whether the test suite is able to find the errors.

Definition 1
A program that is seeded with faults is called a mutant.

Definition 2
If a mutant and the original program produce different outputs on at least one test case, the fault isdiscovered. In this case, we say that the mutant is killed or dead by the test suite.

Let P be the original program, and T be a test suite to test P.

Each inserted fault results in a new program P_1, P_2, P_n called a mutant.

> For $1 \leqslant i \leqslant n$, P_i is called killed if and only if there exists $t \epsilon T$ such that $P_i(t) \neq P_i(t)$ where P(t) is the output ofexecuted P with input t.

Figure 1 illustrates the process of the mutation testing technique.

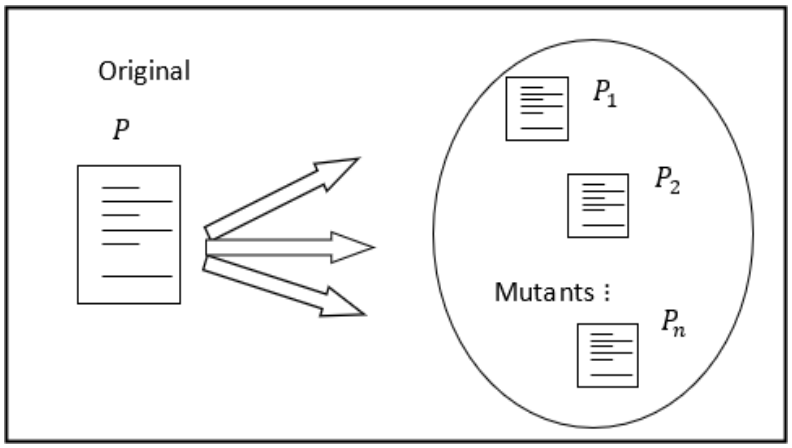


Figure 1: Mutation Testing Process

Definition 3
Given a test suite, if no test case can distinguish between the mutant and the original program, then the mutant is still alive.

If all mutants are killed, then the program is well-tested. If a mutant is still alive, it indicates that the test suite cannot detect the seeded fault. The test suite is said to be inadequate. When the program and a mutant produce the same output for all possible inputs, the mutant is an equivalent mutant.

Definition 4

$$\text{Adequacy of Test Suite} = \frac{\text{Number of killed mutants}}{\text{Number of non} - \text{equivalent mutants}}$$

The ratio of the number of killed mutants to the number of non-equivalent mutants provides a clear idea about the adequacy of a test suite.

Mutation Operators

Using mutation is very important in evaluating, comparing, and improving the quality of a test suite. However, the value of mutation testing depends on the set of mutants used in the evaluation. These mutants are formed from the original program using predefined mutation operators.

Since mutation is always based on mutation operators, researchers have designed and developed mutation operators to support various programming languages, including Java. The quality of mutation operators is the key to mutation testing. One mutation testing used with Java code is MuJava[8, 21]. MuJava includes two general types of mutation operators: class-level operators [22] and method-level operators [8, 23].

Class Level Mutation Operators:

Object-oriented programs differ from traditional programs in many characteristics. They are often structured differently and contain new features such as inheritance, polymorphism, encapsulation, and dynamic binding. Examples of modifying these features by the class mutation operators include deleting the super keyword for inheritance and changing a cast type for the polymorphism.

Several researchers have developed mutation operators for object-oriented software. Kim, Clark, and McDermid [24, 25] proposed a set of class mutation operators for Java. Their operators were developed from the OO Java program features with the intention of representing reasonable flaws. Chevalley [26] has developed additional class mutation operators from OO programs introduced in Java programming language. His class mutation operators are able to generate mutant programs according to 20 kinds of object-oriented Ffaults. Ma, Kwon, and Offut [22] have shown that Chevalley's class mutation operators and other traditional mutation operators are insufficient to handle all OO Java features. Therefore, they proposed a comprehensive set of class mutation operators, including previous traditional Java operators, modifications to previously developed operators, and some new operators, that can handle all Java features and handle several fault types. These operators are those handled by MuJava.

The class mutation operators are divided into four categories: encapsulation, inheritance, polymorphism, and Java-Specific features. These four groups are based on the language features thatare affected [27]. The first three categories depend on programming language features that are common in all OO programming languages. The fourth category depends solely on Java.

Table 1 lists the current set of class mutation operators of MuJava as well as their descriptions.

Table 1: Class Mutation Operators for Java

Language Feature	**Operator**	Description
Encapsulation	AMC	Access modifier change
Inheritance	IHI	Hiding variable insertion
	IHD	Hiding variable deletion
	IOD	Overriding method deletion
	IOP	Overriding method calling position
	IOR	Overriding method rename
	ISI	super keyword insertion
	ISD	super keyword deletion
	IPC	Explicit call of a parent's constructor
Polymorphism	PNC	new method call with child class type
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCD	Type cast operator deletion
	PCC	Cast type change
	PRV	Reference assignment with another comparable variable
	OMR	Overloading method contents replace
	OMD	Overloading method deletion
	OAC	Arguments of overloading method call change
	JTI	this keyword insertion
	JTD	this keyword deletion
	JSI	static modifier insertion
	JSD	static modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Method Level Mutation Operators:

The method level operators, also called traditional operators, include modifications to statements, suchas statement deletion, and modifications to operands and operators, such as arithmetic operation insertion.

MuJava provides six types of method operators [28]:

- Arithmetic operator
- Relational operator
- Conditional operator
- Shift operator
- Logical operator
- Assignment

According to the number and type of operands, some of the method-level operators are divided intotwo or three operators. For example, the arithmetic operator replacement, AOR, is divided into AORB (binary operator), AORU (unary operator), and AORS (short-cut operator). The arithmetic operator insertion, AOI, is divided into AOIU and AOIS. The arithmetic operator deletion, AOD, is divided into AODU and AODS. Table 2 lists the 12 method mutation operators that are used in MuJava.

Table 2: Method Mutation Operators for Java

Kind of Operator	Operator	Description
Arithmetic	AOR	Arithmetic Operator Replacement
	AOI	Arithmetic Operator Insertion
	AOD	Arithmetic Operator Deletion
Relational	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
Logical	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASR	Assignment Operator Replacement

The combined class level and method level mutation operators, including their subdivisions that are currently used in MuJava, are illustrated in Figure 2.

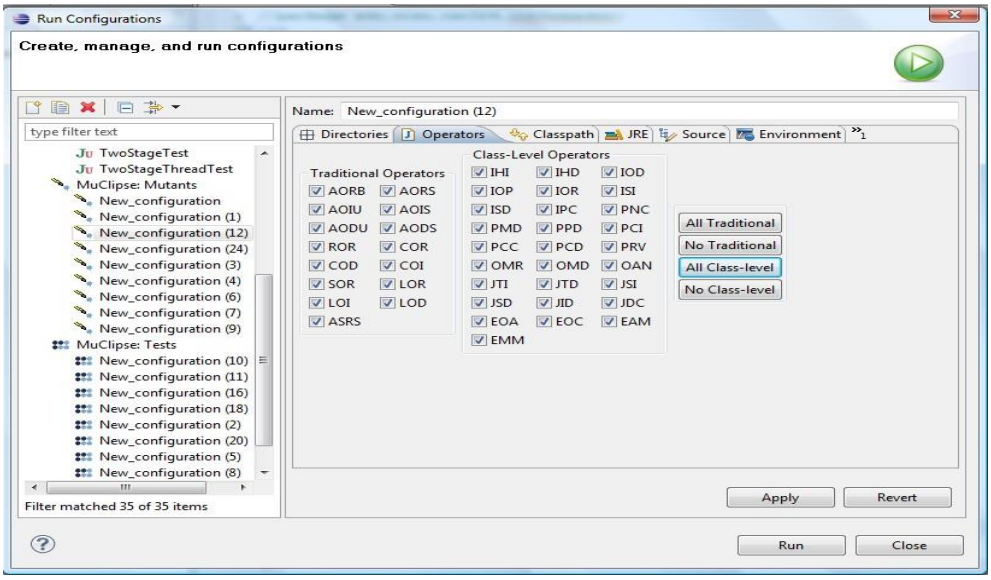


Figure 2: MuJava Mutation Operators

Advantages and Disadvantages of Mutation Testing

Several researchers found that mutation analysis has many attractive features that make it a powerfultool to analyze and automatically enhance a test suite.

Mutation analysis is a technique that is applicable to a variety of programming languages. Since mutation is always based on mutation operators, researchers have designed mutation operators for various programming languages, including Fortran IV, COBOL, C, Lisp, Ada, and Java. In addition, researchers implemented several tools that are developed in different languages to analyze mutationsystems, such as:

- MOTHRA: developed in FORTRAN [23]
- MUJAVA: developed in Java [8, 21]
- PROTEUM: developed in C [29]
- NESTER [24]: a mutation testing tool for C#

Mutation testing provides the tester with a goal. The tester has to develop a test suite that is adequatefor killing all the generated mutants. Hence, the user has the capability to generate an effective test suite that is strong enough to find errors in the program.

Even if no errors are found during testing, mutation still gives the user information about the quality ofthe program tested [30].

Although several empirical studies have shown that mutation testing is highly effective [31, 32], thereare still some issues in using mutation testing.

The massive number of mutants generated is one of the major disadvantages of using mutation testing.Budd determined that the mutants generated for a program are proportional to the number of data references times the number of data objects [33].

It was seen by many researchers that mutation testing is of high computational cost. Even in small and simple programs, software developers spend approximately 40-50% of the development cost and timeto achieve reasonable quality levels [34, 35]. Several techniques were developed to considerably reduce the computational cost and improve effectiveness. Among these techniques are selective mutation [36], mutant sampling [37], weak mutation [38, 39], schema-based mutation [12], and separate compilation [40].

Compiling and executing the faulty versions of the original programs with all the provided test cases isvery consuming. This causes a major problem in using mutation testing. For this reason, mutation testing has not been widely adopted by the software industry [34].

There exists no evidence that the class mutation operators generate realistic faults or that they generate an acceptable number of mutants [41].

Mutation Tool

High-quality software cannot be developed without efficient and effective testing. One commonly used way to evaluate the effectiveness of testing is to use mutation testing [42, 43]. Mutation testingdetermines how effective our test cases are by inserting automatic defects into our program and generating different versions of that program for each defect. Each defect generates a program calleda mutant. Test cases are run against the mutants to detect the inserted defects. If a test case detectsa defect, then the test case kills that mutant. If it does not detect a defect, then the mutant is still alive. If the set of the test cases kills all the mutants, the set is adequate.

MuJava:

The development of mutation tools for object-oriented programming languages has required extensive effort. Many researchers have worked on different techniques seeking to turn mutation testing into a practical testing approach by creating several mutation testings tools. As part of our research, we used a recent and popular testing tool called MuClipse [44]. MuClipse adds a plug-in to the Eclipse platform and is aimed at providing testers with a tool that has an automated generation of mutants as well as running tests against those mutants. MuClipse is based on an implementation of a mutation testing tool called MuJava (μJava) [45] for Java programs.

MuClipse contains two runtime configurations. The first runtime configuration uses a given set of mutation operators on the given original source code to create mutants, which it stores in a folder, usually named results, within the same project, as illustrated in Figure 3.

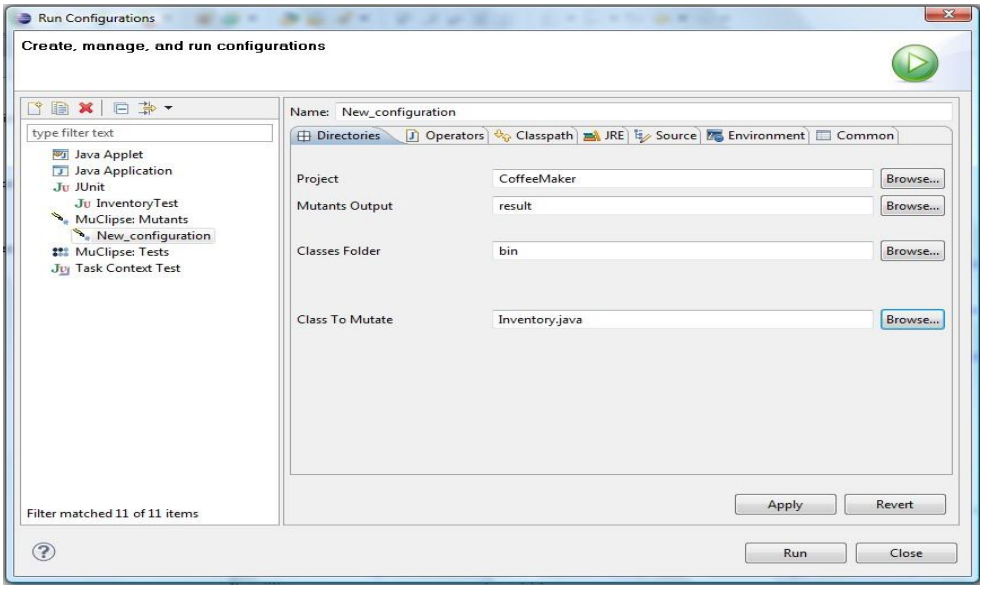


Figure 3: MuJava Run Configuration to Create Mutants

The second configuration runs a given test case or set of test cases on the original code and the generated mutants and compares the results as shown in Figure 4.

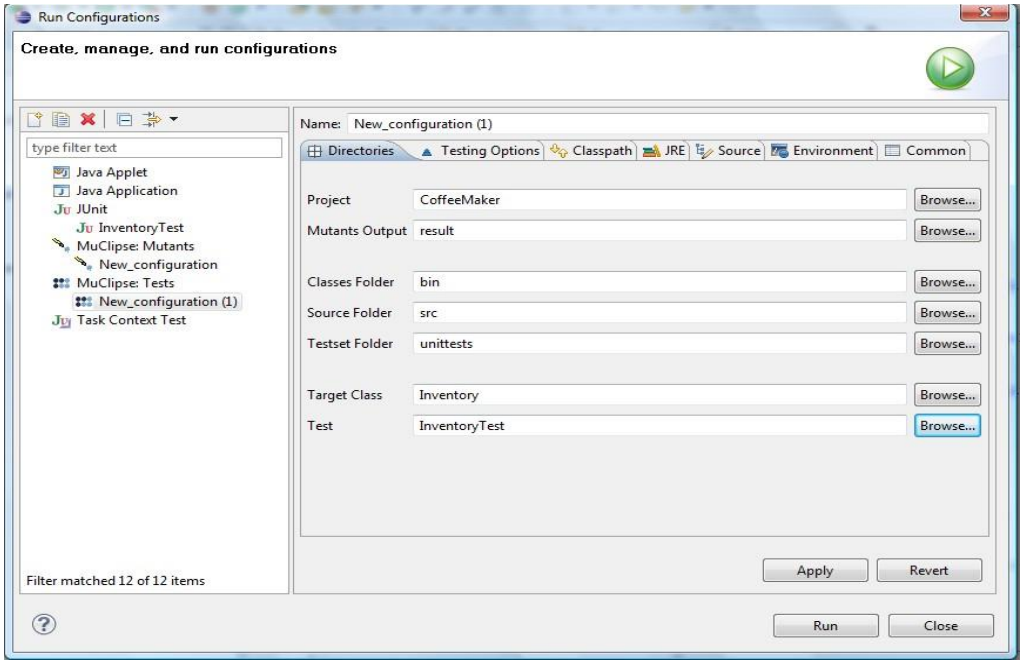


Figure 4: MuClipse Configuration to Run Tests Against Mutants

If no difference between the original program and the generated mutant was detected, the mutant isalive. If the returned result is different, an Exception is thrown, or the mutant causes the JVM to go into an infinite loop, then the mutant is said to be killed.

After all the results have been classified, MuJava returns a ***mutation score*** which is a percentage of allthe generated mutants that have been killed divided by the total number of all the mutants created in the first runtime configuration. Figure 5 illustrates a mutation score of the mutants killed in a Java program. It shows that 92 mutants are killed, and 58 are still alive. These values make the mutant score 61%.

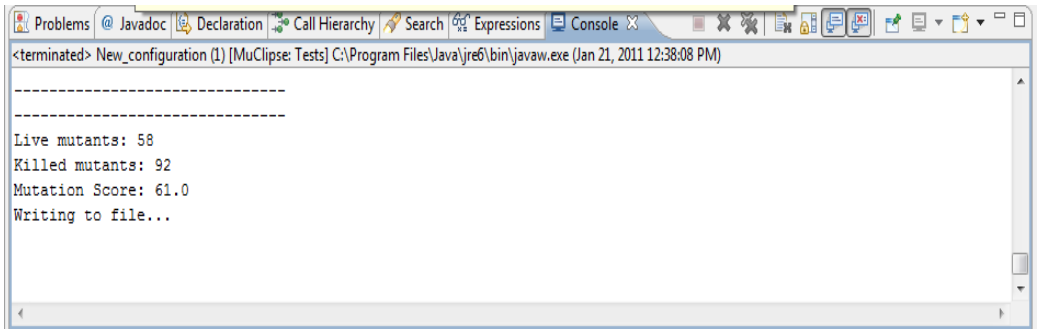


Figure 5: Mutation Score

As discussed in Chapter 2, MuJava includes two general types of mutation operators: class-level operators [22] and method-level operators [8, 23], which are also called traditional mutation operators. Class mutation operators are divided into two types, behavioral mutation operators (operators that change the behavior of the program) and structural mutant operators (operators thatchange the structure of the program). The structure of MuJava makes use of the Mutant Schemata Generation (MSG) approach. Figure 6. illustrates the overall structure of MuJava.

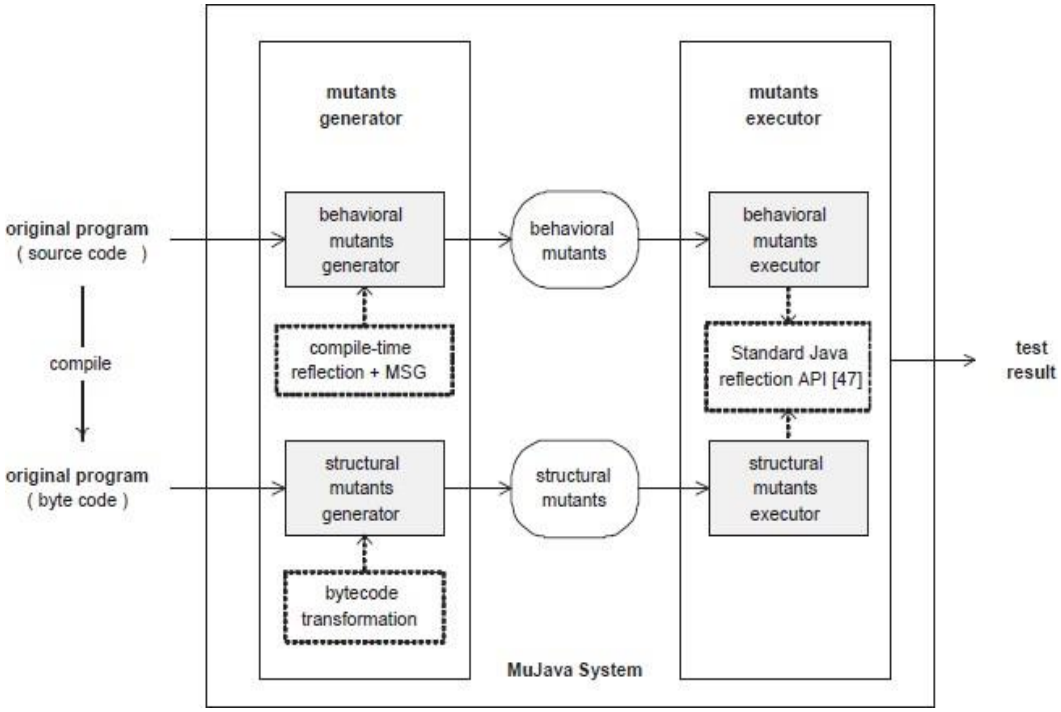


Figure 6: Structure of MuJava [8].

The MSG approach converts all mutants for a program into a program called meta-mutant. It requirestwo compilations: compilation of the original program and compilation of the meta-mutant program.Behavioral and structural mutants are created by different engines. For the behavioral mutants, compile time is used to analyze the original program. The MSG engine then uses the compile-time reflection to create the meta-mutant program. For the structural mutants, the original source code is compiled using the Java compiler. Then, an API is used to add or delete class members in the resultingbyte code transformations [8]. More details on how MuJava works can be found in [8].

Limitations of MuJava:
In our empirical study, we found some limitations in MuJava that made it difficult to execute the mutants of some Java classes and to run test cases against some mutants.

MuJava does not mutate a class that contains a generic type. Generics allow us to abstract over types.For example, the type List<T> is a generic type: a list that holds elements of some type represented by the placeholder T. Figure 7 illustrates the error that is executed when we try to generate the mutants of a class that has a generic type declaration.

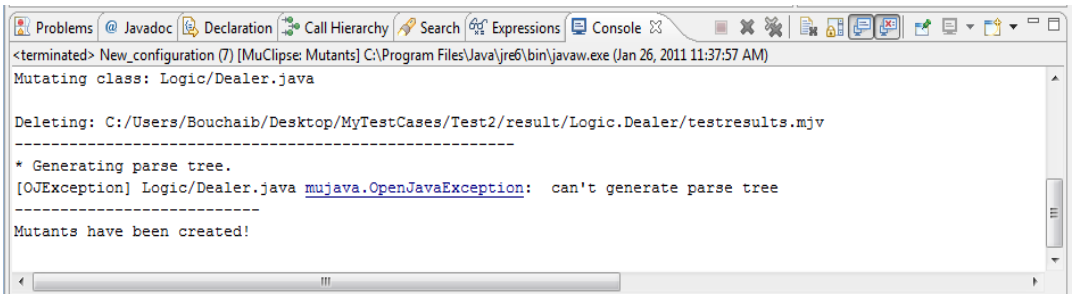


Figure 7: Limitation of MuJava with Generic Type Declarations.

The Logical Operator Insertion (LOI) inserts a negative sign before a variable. When this mutation operator is inserted before a static final variable whose value is zero, the mutated variable does not change at all. Thus, the mutant will never be killed.

Sometimes, when we run the test cases against the mutants, the procedure of running configurationtesting becomes very slow. This is due to the duplication of the operations done by some mutation operators. For example, the arithmetic operators AOIU and LOI perform the same operation. Therefore, we had to kill our process and re-run the Eclipse platform again.
MuJava does not work with abstract classes and interfaces. Thus, we avoid testing these types of classes.

As a result, mutation testing using MuClipse and MuJava is not easy to run in a production environment. Nevertheless, MuJava is an efficient tool to mutate Java programs and run test sets against those mutants. The creators of MuJava are still enhancing its functionality by creating new versions that are free of bugs, and that can overcome all the limitations that can prevent practitionersfrom using this tool in everyday development.

ERROR SEEDING TESTING TECHNIQUES

In the last decades, several formal test methods have been proposed to determine when a program has been adequately tested. Among those methods is the strategy where faults are intentionally inserted into a source code by one person or a group for the purpose of monitoring the rate of detection and removal and estimating the number of faults remaining in the program. This techniqueis called error seeding. This testing process involves deliberately seeding or adding a certain number of falsely generated errors into the source code of a program and test cases are run to detect defects.

The ratio between discovered and actual seeded faults is used to predict the number of remaining undetected, not seeded faults.

Let S be the total number of seeded errors and s be the number of seeded errors found during testing.Let U be the total number of indigenous errors, and u be the number of indigenous errors detected during testing.

The ratio of the number of seeded defects detected to the total number of seeded defects equals theratio of the unseeded defects that have been detected to the total number of unseeded defects [26].

$$\frac{s}{S} = \frac{u}{U}$$

$$U = u \times \frac{S}{s}$$

$$U - u = u \times \frac{S}{s} - u$$

Thus, the number of errors remaining in the system is:

$$U - u = u(\frac{S - s}{s})$$

One of the assumptions of this formula is that the error seeding technique works adequately only if the type of seeded errors matches the type of errors that are originally in the system.

The error seeding concept has been around since at least the early 1970s [46] but has fallen from practice because of difficulties in seeding defects in ways that satisfy the underlying assumptions of the estimation formula. Researchers feel it is worth exploring new approaches to defect seeding, which significantly strengthen the ability to satisfy the underlying assumptions of the estimation formula. Thus, the exploration of mutation testing as a similar resource and alternative strategy was developed in 1978.

Anatomy of Errors

In any project, errors come in three different major shapes: something missing, something wrong, orsomething added. Something missing can be a requirement or a project characteristic that was not implemented. Something wrong can be a feature that was not implemented correctly, including coding mistakes. Something added can be an extra feature outside the project that was created by developers on their own to fill in perceived gaps in requirements. This extra feature can create a holein the testing phase since there will not be test cases for that feature.

Human beings are fallible. Nevertheless, they were and are still essential players in producing software. Errors caused by human beings can come from many different sources. Rushed schedules or time pressure can lead to many mistakes. Lack of training or experience can increase the chance oferrors. Inheriting someone else's code can make it very difficult or impossible for developers to produce bug-free projects. Lack of communication and interaction between developers can generatedefects even before coding starts.

Therefore, developers, testers, and managers have to analyze and be conscious of all defects that cancause a project failure. It is impossible to clean all defects from a program code, but it is possible to avoid defects by not repeating the same mistakes. David Reilly [47] noticed that it is natural that everyhuman being can make mistakes. Based on his experience in Java and the suggestions of many peopleform a group for discussion of Java as a programming language called comp.lang.java. programmer

newsgroup, David Reilly developed a list of the ten top errors that Java programmers usually make, asshowed in table 3 This list will be used in our error seeding experiment.

Table 3: Top Ten Java Errors.

Top ten Errors Java Programmers Make
Accessing non-static members from static methods
Mistyping a methods' name when overriding
Using incorrect comparison assignment (= rather than ==)
Using == rather than. equals when comparing objects
Confusion over passing by value, and passing by reference
Writing blank exception handlers
Forgetting that Java is zero-indexed
Preventing concurrent access to shared variables by threads
Capitalization errors
Null pointers errors

Recently, Bob Martin et al. [51] developed a list of the top 25 most dangerous software errors that can lead to serious software vulnerabilities. This list was developed in three categories:

- Insecure Interaction between Components
- Risky Resource Management
- Porous Defenses

Table 4 lists the software errors that correspond to each category. This list is a resource for programmers, customers, and managers to prevent the types of vulnerabilities that infect the software before it is released to the user.

This list was built based on (1) the original list that was released in 2009; (2) the input from 20 differentorganizations; (3) the feedback of different software security experts and software developers.

In addition to this list, Bob Martin et al. [51] also provide different steps that developers can follow toprevent these errors, thereby limiting the weakness of the software.

Table 4: 2010 SWE/SANS Top 25 Software Errors by Category

Software Error Category	Software Error
Insecure Interaction Between Components	Failure to preserve web base structure
	Failure to preserve SQL query structure.
	Cross-Site request forgery
	Unrestricted upload of file with dangerous type
	Failure to preserve OS command structure.
	Information exposure through an error message
	URL redirection to untrusted site
	Race condition
Risky Resource Management	Buffer copy without checking size of input.
	Improper limitation of a pathname to a restricted directory
	Improper control of filename for include/require statement in PHP program buffer access with incorrect length value
	Improper check for unusual or exceptional conditions
	Improper validation of array index
	Integer overflow or wraparound
	Incorrect calculation of buffer size
	Download of code without integrity check
	Allocation of resources without limits or throttling
	Buffer Access with Incorrect Length Value
Porous Defenses	Improper Access Control (Authorization)
	Reliance on Untrusted Inputs in a Security Decision
	Missing Encryption of Sensitive Data
	Use of Hard-coded Credentials
	Missing Authentication for Critical Function
	Incorrect Permission Assignment for Critical Resource
	Use of a Broken or Risky Cryptographic Algorithm

COMPARISON OF MUTATION AND ERROR SEEDING TECHNIQUES

Both mutation testing and error seeding intentionally seed errors into a program under test to verify that the test can detect the error. However, they differ in many points. Table 5 lists the differences between mutation testing and error seeding techniques.

Table 5: Comparison of Error Seeding and Mutation Testing

Error Seeding	Mutation Testing
Mutants do not exist here.	Mutants are created for testing.
Source code is tested within itself.	Mutants are combined, compared for testing to find inserted errors.
Errors are introduced directly.	Special techniques are used to introduce errors.
Test cases which detect errors are used for testing.	Test cases which kill mutants are used for testing.
Less time is required.	More time consuming.
Economical technique.	Expensive technique.

To evaluate our approach, we have conducted a series of experiments based on three open-source applications, CruiseControl, Blackjack, and CoffeeMaker downloaded from three different websites[48, 49, 50]. The evaluation of our approach was done by calculating:

- The ratio of killed mutants by the total number of killed and alive mutants for mutation technique.
- The ratio of the number of killed seeded errors to the total number of detected and non-detected errors for the errors seeding technique.

Tables 6 and 7 show both ratios.

Table 6: Rate of Detection of Non-Detected Errors.

Application	Class		Unit Testing			
	Name	# of methods	# of tests	Killed mutants	Live mutants	Score
Blackjack	Dealer	1	3	13	0	100%
	Hand	3	4	32	0	100%
	User	5	6	34	0	100%
CoffeeMaker	CoffeeMaker	8	12	92	60	61%
	Inventory	10	16	100	50	66%
CruiseControl	CarSimulator	11	13	75	20	80%
	SpeedControl	7	8	60	0	100%

Table 7: Rate of Detection of Non-Detected Errors.

Application	Class		Unit Testing			
	Name	# of methods	# of tests	Detected Seeded Errors	Non-Detected Seeded Errors	Rate of Detection
Blackjack	Dealer	1	2	8	0	100%
	Hand	3	4	3	0	100%
	User	5	6	5	0	100%
CoffeeMaker	CoffeeMaker	8	9	3	0	100%
	Inventory	10	11	3	0	100%
Cruise Control	CarSimulator	11	12	14	0	100%
	SpeedControl	7	8	4	0	100%

Both tables show that the fewer the test cases are, the higher the ratios are and, hence, the more effective our testing strategy is. They also show that, while all seeded errors were detected, the number of mutants that are still alive is very high in the methods with more test cases.

CONCLUSION

Test regression selection is a critical aspect of software testing that aims to identify a subset of test cases from an existing test suite to ensure that software modifications have not introduced any new bugs. In this study.

Mutation testing is a fault-based testing technique used to find the effectiveness of test cases and assure the quality of software. Mutation testing is an automated technique that involves introducing small artificial faults, known as mutations, into the codebase and then executing the existing test suiteagainst these mutated versions. The purpose is to evaluate the effectiveness of the test suite in detecting the introduced faults. If a test case fails to detect a mutation, it indicates an inadequate testcase.

On the other hand, manual seeding testing is a technique where faults are intentionally inserted intoa source code by one person or a group for the purpose of monitoring the rate of detection and removal and estimating the number of faults remaining in the program.

This study compares two techniques for regression selection: Mutation testing and Manual Seeding testing. This comparison was in terms of their advantages and disadvantages. It is conducted throughan empirical evaluation of a set of three real-world open-source application projects.

The evaluation includes measuring the ability of the techniques to detect regressions accurately and comparing the ratio of killed mutants by the total number of killed and alive mutants against the ratioof the number of killed seeded errors to the total number of detected and non-detected errors.

The findings of this study provide insights into the strengths and weaknesses of mutation testing and manual seeding testing techniques for test regression selection. The results contribute to informed decision-making regarding selecting appropriate techniques based on project requirements andconstraints.

The main difference between the two techniques lies in their intended scope and purpose. Mutationtesting primarily serves as a rigorous method to assess the quality of test cases and evaluate their fault-detection capabilities. On the other hand, seeding errors might be used more informally by developers during their individual work to gain insights into specific parts of the code or to improve their own understanding of the system's behavior under error conditions. While both techniques offer valuable insights into the software's resilience and test suite quality, mutation testing stands out as a more formal and systematic approach adopted by researchers to assess the overall effectiveness of test cases and ensure robust software performance. However, both techniques have their place in the development process, complementing each other to produce reliable and stable software products.

References

[1]. R. A. DeMillo, R. J. Lipton, and F. J. Sayward, "Hints on Test Data Selection: Help for the Practicing," Computer, Vol. 17, No. 4, pp. 34-41, April 1978.

[2]. A. T. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward, "Mutation Analysis," Research Report # 155, April 1979.

[3]. A. Timothy Budd, "Mutation Analysis of Program Test Data," PhD Thesis, pp.155, Yale University.

[4]. A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," Proc. Mutation, San Jose, CA, USA, pp. 45- 55, October, 2000.

[5]. T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," in Proceedings National Computer Conference, AFIPS Conference Record, pp. 623-627.

[6]. A. J. Offutt and K. N. King, "A Fortran 77 Interpreter for Mutation Analysis," in 1987 symposium on Interpreters and Interpretive Techniques, (ST. Paul MN), pp. 177-188, ACM SIGPLAN.

[7]. R.A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King, "An Extended Overview of the Mothra Software TestingEnvironment," in Proceedings of the 2nd workshop on Software Testing, Verification, and Analysis, pp. 142-151, IEEE Computer Society Press.

[8]. Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon, "MuJava: An Automated Class Mutation System," Software Testing, Verification and Reliability, 2005, pages. 97-133.

[9]. A. J. Offutt and R. A. DeMillo, "Constraint-Based Automatic Test Data Generation," 1EEE Transactions on Software Engineering, Vol. 17, no. 9.

[10]. T. Denmat, A. Gotlieb, and M. Ducasse, "Improving Constraint-Based Testing with Dynamic Linear Relaxations," In 18th IEEE International Symposium on Software reliability Engineering (ISSRE' 2007), Trollhttan, Sweden, Nov. 2007.

[11]. A. Jefferson Offutt, "An Integrated Test Data Generation System," Journal of Systems Integration, Volume.1, pp. 391- 409.

[12]. A. J. Offutt, R. H. Untch, and M. J. Harrold, "Mutation Analysis Using Mutant Schemata," in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA' 93), Cambridge, Mass, ACM SIGSOFT SE Notes, Vol. 18, No. 3, 139-148.

[13]. T. L Graves, M. J. Harrold, J. Kim, A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Selection Techniques," ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 2, April 2001, Pages 184-208.

[14]. Jun-Ming Kim and Adam Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments," International Conference on Software Engineering, Proceedings of the 24th International Conference, 2002.

[15]. Jun-Ming Kim, Adam Porter, and G. Rothermel. "An Empirical Study of Regression Test Application Frequency," In Proc. of the 22nd International Conference on Software Engineering, pages 126-135, June 2000.

[16]. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Control-flow-Based Test Adequacy Criteria," In Proc. of the 22nd International Conference on Software Engineering, pages 191-200, May 1994, IEEE.

[17]. D. Rosenblum and G. Rothermel, "A Comparative Study of Regression Test Selection Techniques," In Proc. of the 2nd International Workshop on Empirical Studies of Software Maintenance, October 1997.

[18]. G. Rothermel and M.J. Harrold, "Aristotle: A System for Research on and Development of Program-Analysis-Based Tools," Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, March 1997.

[19]. G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," ACM Transactions on Software Engineering and Methodology, Vol. 6, Issue 2, pages: 173 – 210.

[20]. G. Rothermel and M. J. Harrold, "Empirical Studies of a Safe Regression Test Selection Technique," IEEE Transactions on Software Engineering, Vol. 24, No. 6.

[21]. A. J. Offutt, Y. S. Ma, and Y. R. Kwon, "An Experimental Mutation System for Java," ACM SIGSOFT Software Engineering Notes (SECTION: Workshop on empirical research in software testing papers), 29(5), Sep. 2004.

[22]. Y. S. Ma, Y. R. Kwon, and J. Offutt, "Inter-Class Mutation Operators for Java," In Proc. of ISSRE 2002, pages 352–363, IEEE Computer Society Press, Nov. 2002.

[23]. A. Jefferson Offutt and K. N. King, "A Fortran Language System for Mutation-Based Software Testing," Software Practice and Experience, 21(7):686–718, July 1991.

[24]. S. Kim, J. Clark, and J. McDermid, "Assessing Test Set Adequacy for Object Oriented Programs Using Class Mutation," 28 JAIIO: Symposium on Software Technology.

[25]. S. Kim, J. Clark, and J. McDermid, "Class mutation: Mutation Testing for Object-Oriented Programs," OOSS: Object- Oriented Software Systems, October 2000.

[26]. Steve McConnell, "Gauging Software Readiness with Defect Tracking," IEEE Software, Vol. 14, No. 3, May/June 1997

[27]. Y. S. Ma and J. Offutt, "Description of Class-level Mutation Operators for Java," November 7, 2005.

[28]. Y. S. Ma and J. Offutt, "Description of Method-level Mutation Operators for Java," November 29, 2005.

[29]. M. E. Delamaro and J. C. Maldonado, "Proteum - A Tool for the Assessment of Test Adequacy for C Programs," In Proceedings of the Conference on Perform ability in Computing Systems, pages 79–95, East Brusnwick, New Jersey, 1996.

[30]. B. Baudry, V. L. Hanh, J. M. Jézéquel, and Y. L. Traon, "Trustable Components: Yet Another Mutation-Based Approach," in Proceedings of the Symposium on Mutation Testing for the new Century(Mutation 2000), pages 69–76, San Jose, California, October 2000.

[31]. D. Schuler, V. Dallmeier, and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," In ISSTA '09: Proceedings of the 18th International, Symposium on Software Testing and Analysis, pp. 69–80, New York, NY, USA, 2009, ACM.

[32]. J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? In ICSE '05: Proceedings of the 27th International Conference on Software Engineering, pp. 402-411, New York, NY, USA, 2005. ACM.

[33]. A. T. Budd, "Mutation Analysis: Ideas, Examples, Problems and Prospects," Computer Program Testing, pages 129–148.

[34]. S. Madiraju, S. Ramakrishnan, and A. J. Hurst "Towards Automated Mutation Testing," March 2004.

[35]. John J. Marciniak, "Encyclopedia of Software Engineering," Wiley- Interscience.

[36]. L. Bottaci, E. S. Mresa, "Efficiency of Mutation Operators and Selective Mutation Strategies". An empirical study, Software Testing, Verification and Reliability, pp. 205 - 232.

[37]. T.A. Budd, "Mutation Analysis: Ideas, Examples, Problems and Prospects," In Proceedings of the Summer School on Computer Program Testing, Sogesta, pp. 129 - 148.

[38]. W.E. Howden, "Weak mutation testing and completeness of test sets," IEEE Trans- actions on Software Engineering, pp. 371-379.

[39]. M. R. Woodward and K. Halewood, "From Weak to Strong, Dead or Alive? An Analysis of some Mutation Testing Issues," In Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Canada.

[40]. C. Byoungju and A. P. Mathur, "High-Performance Mutation Testing," The Journal of Systems and Software, pages 135-152.

[41]. Y. S. Ma, M. J. Harrold, and Y. R. Kwon, "Evaluation of Mutation Testing for Object Oriented Programs," ICSE'06, Shanghai, China,ACM 159593085X/ 06/0005, May 2006.

[42]. R. G. Hamlet, "Testing Programs with the Aid of a Compiler," IEEE Transactions on Software Engineering, 279 – 290.

[43]. K. Adamopoulos, M. Harman, and R. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-Evolution," IN GECCO (2), Volume 3103, 2004.

[44]. "Muclipse: An Open-Source Mutation Testing Plug-in for Eclipse," July 2008, http://muclipse.sourceforge.net.

[45]. J. Offutt, "MuJava Home Page," Assistance from the U.S. National Science Foundation under Grant CCR-98-04111, November 2008, http://cs.gmu.edu/~offutt/mujava.

[46]. B. Boehm and D. Port, "*Defect and Fault Seeding In Dependability Benchmarking*," In Proc. of the DSN Workshop on Dependability Benchmarking, June 2002.

[47]. D. Railey, "Top Ten Errors That Java Programmers Make," June 2006, http://www.javacoffeebreak.com/articles/toptenerrors.html.

[48]. James Hahn, "Sourceforge: Simple Swing Java Blackjack," 2009, http://sourceforge.net/projects/simpleswing21.

[49]. Laurie Williams, Dright Ho, and Sarah Heckman, "CoffeeMaker Example," http://agile.csc.ncsu.edu/SEMaterials/tutorials/coffee_maker, Last updated: March 2009.

[50]. Software-artifact Infrastructure Repository, "Cruise Control," 2009, http://sir.unl.edu/content/bios/CruiseControl.html.

[51]. B. Martin, M. Brown, A. Paller, and D. Kirby, "2010 CWE/SANS Top 25 Most Dangerous Software Errors," December 2010, http://cwe.mitre.org/top25/index.html.