UNIVERSITY OF OSLO
Department of Informatics

Towards Safe Mutation Testing in a Sandbox Environment

Master of Science Thesis

Ronny Mandal

May 2, 2011

Abstract

Mutation Testing (MT) is a technique for evaluating how well software is tested. MT makes small changes to the software, and the goal is to see whether the current test cases are able to distinguish *mutants* from the original software. If mutants are not distinguished, it is likely that the software was not tested well enough. However, apart from trivial software, making changes to software might have dangerous side effects on the host where test cases are executed. For example, a program that manipulates files might end up in deleting or overwriting important files in the file system if such program is arbitrarily mutated with MT. For programs written in Java, it is possible to execute MT in a *sandbox*, to avoid these types of problems. But how often such problems happen in practice? What is the overhead of using such a sandbox? Are there ways to improve MT to reduce the negative impacts of these side effects? In this thesis, we investigate whether and how often mutants cause undesirable side effects. We carried out MT sessions for ten different large real world projects downloaded from SourceForge, and wrote tools to analyze the results and run MT in a sandbox. The data from these experiments are used to study several correlations among the factors that affect MT applied to real world software where unwanted side effects of the testing phase can be harmful. We identified some types of MT operators that have higher probabiltiy of causing harmful side-effects. These operators could be removed from MT analyzes and tools.

Acknowledgements

Thanks to: Andrea Arcuri for supervision, Lionel Claude Briand for being the lead scientist of the APPROVE group, my family, Jo Schlytter, Helene Skancke and the statisticians at Simula Research Laboratory.

Contents

1 Introduction **11**
- 1.1 Background 11
- 1.2 Mutation Testing 12
  - 1.2.1 Mutant generation 13
  - 1.2.2 Mutation execution 13
  - 1.2.3 Result analysis 14
- 1.3 Problem Description 14
  - 1.3.1 Rationale for this thesis 15
- 1.4 Research Method 15
  - 1.4.1 The JUnit framework 16
  - 1.4.2 Assessment framework 16
- 1.5 Thesis Structure 20

2 Security Exception Sources **21**
- 2.1 The Java Security Model 21
  - 2.1.1 Permissions 21
  - 2.1.2 Policies 22
- 2.2 Native Security Manager - java.lang.SecurityManager 23
- 2.3 The Custom Security Manager 23
- 2.4 Automatic Creation of Policy Files 23
- 2.5 Unstable Host Computer 24
- 2.6 AccessControlException 25
  - 2.6.1 FilePermission 25
  - 2.6.2 ManagementPermission 29
  - 2.6.3 RuntimePermission 30
  - 2.6.4 SocketPermission 31
- 2.7 FilePermission violation 32
  - 2.7.1 Apache Scout - EAM 32
  - 2.7.2 PDFBox - AOIS 33
  - 2.7.3 PDFBox - JSI 34
  - 2.7.4 Maven3 - JTI 36

3 Results and Analysis **39**
- 3.1 Test Subjects and Characteristics 39
- 3.2 Test Subject Metrics and Statistics 40
  - 3.2.1 Preliminary Data Analysis - Code Coverage for projects 2 and 5 42
- 3.3 Research Questions 43
  - 3.3.1 RQ1 45
  - 3.3.2 RQ2 46

3.3.3 RQ3 50
3.4 Mutant Operators and Security Exception Distribution 53
3.4.1 Mutation Operators with a Higher Probability of causing Side Effects . . 55

4 Discussion 57
4.1 Mutants and Evidence of Hazards 57
4.2 Custom Security Manager and Application Domains 58

5 Threats to Validity 61
5.1 Realism 61
5.1.1 Realistic Tasks 61
5.1.2 Realistic Subjects 62
5.1.3 Realistic Environment 62
5.2 Validity 62
5.2.1 Internal Validity 62
5.2.2 External Validity 63
5.2.3 Construct Validity 63

6 Implementation: Challenges and Obstacles 65
6.1 Execution Platform and Storage Capacity 65
6.2 Execution Platform and Processing Capacity 65
6.3 Choice of execution platform 66
6.4 Impediments related to the Execution Process 66
6.4.1 Sub Processes and Separate Memory Spaces 68
6.5 Impediments related to Omissions of the Data Foundation Specification 68

7 Conclusion 69

A Appendices 73
A.1 Why not Byte Code Mutation? 73
A.2 CSM Performance 76
A.3 Critical Values of Correlation Coefficient (R) 77
A.4 Chi-square Distribution Table 78

List of Figures

1.1 Java Runtime System Diagram 15
1.2 Abstract Overview over the Framework 17
1.3 Java Runtime System Diagram with security manager 17
1.4 Book Keeping of Class Files 18
1.5 Byte code tree 19
1.6 Mutant tree 20

2.1 The Java Security Model 22
2.2 An Error Message caused by a missing File 24
2.3 Kernel Panic on Linux/FreeBSD 25

3.1 Test Code Coverage with failed Fests 43
3.2 State Machine Diagram of the Result Evaluation Process 44
3.3 Proportions for Results 45
3.4 The Class-Mutant Relationship for RQ2 46
3.5 Test Cases passed per Class 49
3.6 Test Cases failed per Class 49
3.7 Test Cases with Security Exception per Class 49
3.8 Test Cases with Security Exceptions per Class w/o Outliers 49
3.9 The Mutation Operator-Class Relationship for RQ3 51
3.10 Test Cases passed per Mutation 52
3.11 Test Cases failed per Mutation 52
3.12 Test Cases with Security Exception per Mutation Operator 52
3.13 Test Cases with Security Exception per Mutation Operator w/o Outliers 52
3.14 Security Exception Distribution 53

6.1 Unix Pipe 68

A.1 Execution Time for the CSM 76

List of Tables

3.1 Project with Indices for Reference 40
3.2 Data from the Mutation Testing Process 41
3.3 Code Coverage Details 41
3.4 Kruskal-Wallis Test for Outcome per Class 48
3.5 Kruskal-Wallis for Mutated Classes 51

6.1 Mutants and their Execution Time 66

Chapter 1

Introduction

Our society is becoming more and more dependent on computers. The evidence is the abundance of computer controlled devices (CCD) that surrounds us. While these devices make our lives more convenient as we rely heavily on them, the quality of service (QOS) in the sense of normal operation is inevitably becoming crucial. In the ideal world, every CCD should be error-free. However, it is not feasible to produce software without errors. [15, 32].

The infeasibility of error-free software imposes us to endure with "good enough quality" of software [8, 12, 22, 57]. This necessarily applies to our CCD also as they are software controlled. The idea of "good enough quality" is connected to the cost of the testing process, which is a substantial part of the whole software development process. The extent of the testing process, i.e. when to finish, is related to the quality requirements of the software. This means that important software undergoes a more thorough testing process than less important software as the consequences of a malfunction weighs more in the case of the first.

Nevertheless, it makes sense to improve the QOS without a substantial increase in the total cost of the software [17]. To realize this idea, the approach of test automation (among others) have been suggested. There are a plethora of different test automation methods. Mutation testing is a considerable contribution to the test automation paradigm.

Mutation testing is a *fault-based* testing technique. Testing is fault-based when its motivation is to demonstrate the absence of prespecified faults. The main idea is to introduce faults into correct programs to produce faulty versions. These faulty versions are variants of the original and is referred to as a mutant. These faults can be seeded manually by an experienced programmer, or it can be done automatically. When it is performed in the latter fashion, the mutant is generally viewed as the result of applying an operator which transforms the code. The process of analyzing when mutants fail and test suites trigger is referred to as mutation analysis.

This testing technique was initially proposed by Lipton [14] and Hamlet [24]. Since then, the development of mutation testing has evolved [30]. A lot of research is conducted with the objective of refining the mutation testing process, e.g. optimization of resource utilization [46, 47, 59, 61], elimination of mutation operators which generates multiple variations of the same mutated statement [64] and removal of mutants which cannot be detected by a test suite. [23,58]. We strongly believe that further improvements of the mutation testing process is welcomed by the testing community.

1.1 Background

The premise for a well accomplished development project is a framework to structure, plan and control the process of developing the final product, i.e. the information system. This framework is usually based on formalized methodologies or deviations of the latter. We will refer to this as the development model. This development model is divided into segments, where the former segment is a prerequisite for the next. These segments will be referred to as phases. Likewise, test engineers are using methodologies to organise the quality assurance (QA) tasks. This is often referred to as the QA process. The QA process is ideally adapted to the development model allowing QA to co-exist with the development process. The V-Model [9] is an example of a model for a QA process.

The resource consumption of the QA process tends to vary in extent, sometimes occupying up to 80 percent of the whole development cost [4, 55]. This is justified on the grounds that the cost of correcting an error increases exponentially when it is allowed to live over a transition of a phase in the software development process. [68].

To alleviate the cost of the process, one aims to move as much of the test process as possible to the automated space. [26]. However, as a consequence of this new challenges emerge, as for mutation testing.

1.2 Mutation Testing

A *fault based* [43] approach to test automation is *mutation testing*, seminally described by Lipton [14] and later elaborated by, inter alia, A.J. Offutt. Testing is *fault-based* when its motivation is to demonstrate the absence of prespecified faults. These faults may be introduced manually, preferably by experienced programmers, or it may be generated automatically by lexical analyzing the code and apply them by following a predefined pattern. This is called *mutant generation.*

The main advantage of this technique is that the faults are described precisely and thus provide a well-defined fault-seeding process, as opposed to manual seeding. Every faulty program is then executed and the results are logged. The last, but important step is to analyze when these mutated programs fail. This is known as *mutation analysis.* In this thesis, we refer to test cases without any mutation operator applied to its encompassed classes to the *pre-mutated test case.* When a mutation operator is applied to at most one of its classes, we refer to the test case as the *mutated test case.* Sometimes it is referred to as simply the *test case* and the status, i.e. mutated or not depends on the context.

Given a program p, a test case τ, mutation operators $\mu_n, n \in \mathbb{N}$ and a test oracle O, the idea behind mutation testing is basically to produce small variants, e.g. p_1, p_2, p_3 by applying μ_1, μ_2, μ_3 to p [49]. An application of a μ is similar to an error in code done by a programmer [7]. When the pre-mutated $\tau(p)$ is run, the output should comply to the description given by *O(τ).*

After an execution of $\tau(p_1)$ is done, its output (ideally) is distinguished or is not distinguished from *O(τ).* If the difference between $\tau(p)$ and $\tau(p_n)$ is detected, we say that μ_n is killed. If the difference is undetected, μ_n is referred to as a *live mutant.* The more mutants that are killed by τ, more likely is it that τ will detect errors similar to the killed mutants.

When a mutant is live, it is either an *equivalent mutant* or the portion of the class where this mutation reclines is not exercised, i.e. not covered by the test. A third option is that it only caused *weak mutation* [27] to the code. Weak mutations does not propagate through the execution cycle, thus does not affect the outcome. An equivalent mutant causes grammatical modifications to the code, but does not modify its semantics [23] with respect to the input. For this reason, it remains undetected.

This method of mutation testing has proven well in assessing the robustness of test cases [19], despite that it also contains weaknesses [23, 58]. Coarsely, mutation testing consists of three steps, mutant generation, mutant execution and result analysis. All these steps are expensive both computationally and manually. Expensive in the means of the latter because analyzing requires manual labour. We will understand why it is expensive w.r.t. both fashions after reading the next three subsections and understand that a fully automated and unattended process of mutant generation and mutant execution is crucial.

1.2.1 Mutant generation

The first step is to analyze the source code to establish which mutants that are eligible to the code. This process includes a lexical analysis of the source code in conjunction with a set of rules describing which modifications that can be done without violating the grammar of the language. The modification or *mutant patterns* are described by *mutation operator*, such as e.g. *AOI* and *JTD*. These two replaces an arithmetic operator (e.g. it replaces a '-' with a '+') and removes a *this*-keyword (*this.foo* = *foo*) respectively. Please refer to the bibliography [38, 51, 52] for more information on these operators. In the context of Java, for every class for which a mutation operator is applied, an extra class file is produced. In our study, a mutated class will contain at most one modification produced by applying a mutation operator, except from the example in Section 3.4.

In this experiment, two different classes of mutation operators are applied to the code; *traditional mutation operators* and *object-oriented mutation operators*. The aforementioned mutation operators are examples of mutation operators belonging to these classes respectively. Traditional mutants makes modifications to the methods in code [37], while object-oriented mutants makes modifications to object oriented code constructs, e.g. encapsulation, inheritance, and polymorphism [52].

Papers report that thousands of mutants were generated from a relatively few lines of code (LOC). From [50] we learn that by analyzing mutant generation from 28 Fortran-77 programs with LOC ranging from 8 to 164, 43 to 27331 mutants were produced and a total of 81159 accumulated mutants. [44] reports a mean of 3211 mutants generated from 10 programs with a mean of 43.7 source statements. Dasso et al. suggests the numbers of mutants generated to be the square of the LOC [3, p. 136]. Effectively, this means that **one** compilation is required for each mutation operator applied to the source code. Please be cognizant of that compilation implicates all necessary classes and libraries to be loaded for each and every compilation. When the amount of mutants applied is large (which usually is the case), this process is computationally expensive.

1.2.2 Mutation execution

Mutation testing is often performed in conjunction with unit testing, which also is the case for this experiment. When performing mutation testing in this context, the total execution time is roughly determined by the execution time of the pre-mutated test case multiplied by the numbers of executions required to execute all generated mutants encompassed by that test case. In Java, this typically means that if a test case consumes t time units and n mutants are eligible to the classes in the test case, the total execution time will increase by approx. $t \times n$ time units.

In advance, it is difficult to predict outcome w.r.t. execution time of a test case involving a mutated class. An estimated execution time can for simplicity be expressed as:

O($t \times n$), t is the time to run the pre-mutated test case, n is the number of mutated classes encompassed by the test case.

Some of the executions of the same test case may take longer than the pre-mutated test case or it may terminate prematurely due to exceptions or errors. According to Table 6.1, this will level off, hence we justify the average time approximately equal to t.

When this is being performed with the number of mutant expected (we expected a vast amount of mutants), the total execution time may take days or weeks. In our study, we ran 12 projects (which two were rejected due to inadequate results) over a period of 5 weeks.[1] No need to further elaborate that this is an expensive process.

1.2.3 Result analysis

The mutants that are killed, normally requires no further extensive analysis, as opposed to the live mutants. In a study conducted by Schuler et al., live mutants were dichotomized into *"not covered"* and *"covered, not killed"* [23] with a distribution of 32 and 20 percent respectively. The first group suggests that a significant amount of the mutants in our study remain unassessed. The second group was inspected manually, revealing that 40 percent of the mutants were considered to be equivalent. The same authors also reports that the average time consumption for manually assessing a mutant is 15 minutes [58].

In this study, however, we are not concerned by the analyze of live mutants. Our principal interest are the mutants which cause file access violations. When such an incident are detected, they are of course being detected.

1.3 Problem Description

With the aforementioned characteristics of mutation testing in mind, there is no need to say that it is desirable that such a process should, between commencing and completion, persist unattended. An unexpected premature termination of this process could, in the full extent, havoc a project considering the strict constraints that generally confine the software development projects to a given time frame.

Considering Java, a run-time environment is required for program execution, namely the Java Runtime Environment (JRE). The JRE can be visualized as a logical layer between the operating system (OS) and the Java application (Figure 1.1). The JRE constitutes a virtual machine (VM) which, in addition to perform execution of the Java byte code, is responsible of loading all required classes, transfer control from the application to the OS (file requests, network calls etc). When, for instance the application requests access to a file residing in the file system of the OS, it is in fact the JVM that accesses this file on behalf of the application. Even though this is transparent to the programmer, a lot of system calls are being performed "behind the scenes".

From the introduction we know that no computer program is error free. A programmer may unintentionally write code that performs deletion of vital files, thus causing an impact on the availability of the resources of the OS. Since we also know that mutation operators emulates error made by programmers, it is reasonable to assume that mutation operators may modify the code so that the program renders the host computer unstable by intervening with the run time system.

"Out of the box" JVMs does not enforce any restrictions on which resource that can be accessed, hence a hostile application may operate on any file available, including those which are vital for the OS. (This does not apply when running *Java-Applets* where the security is enabled by default.) The consequences of a malicious file deletion could be severe. The aftermath of such an operation may demand an audition of the logs by systems engineer to reconstruct the

[1]We needed to restart the process occasionally, when this is taken in account, the whole process elapsed for almost 10 weeks.

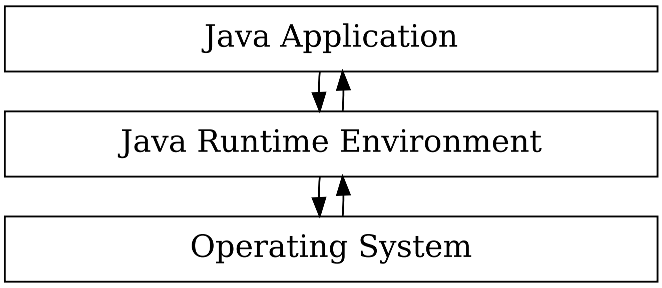


Figure 1.1: An abstract and simplified overview over the Java Runtime Environment (JRE). The arrows shows transfer of control.

run-time environment of the OS or other files that may have been inappropriate deleted. It is also possible that such an incident entails a complete reconstruction of the whole testing environment, which would cause additional delay of the QA process. In addition, the test engineers are required to weed out the mutant causing this incident and re-assess the risk analysis as there might be other possible hazards in the pipeline.

Besides all these tasks, the test environment might be subject to re-design to make it more resilient, resulting in an environment *too different* from the production environment. This raises concerns, as these characteristics may have serious impact on the stability of the QA process [63], thus the release of the final product.

To pave our path towards safe testing and eliminate potential risk factors such as the aforementioned, we want to investigate this claim by analyzing several execution cycles of mutation testing performed on different software projects with the aim to support or abandon the hypothesis of hazardous mutation operators.

1.3.1 Rationale for this thesis

To our best knowledge after reviewing several articles from IEEE and ACM, none of these discuss the potential hazardous side effects of the application of mutations. [6,10,16,18–21,23, 30,31,33,35,47,48,54,58,60,61,64–66,72].

On this basis we want to investigate if evidence of such hazards exists or not. If such a hazard do exists, we also want to discuss usages for the security mechanisms of in the context of mutation testing.

1.4 Research Method

The initial step was to find suitable software projects candidates to take the role of the test subjects or systems under test (SUTs). We wanted to assess code that met certain quality criteria. Firstly it should be open source software. This allows us to inspect code without violate any copyright, let alone inspect the code at all. Secondly, it should be in production and widely utilized or have a life cycle history. This allows us to assume that the source code is/has been subject to evolution and refinement, or at least have a certain maturity making it a suitable test subject. Thirdly it should be easy to install and execute. For this reason, we chose Maven enabled projects. Maven will also provide commands for extracting the class path containing all program dependencies for later use by the assessment framework developed for this purpose (see Section 1.4.2). Fourthly, it must ship with a test suite consisting of JUnit test cases. Our assessment framework supports JUnit only. With these criteria met, the first step of the process, i.e. the mutant generation can begin.

As a mutant generator system, we chose MuJava (formerly JMutation). MuJava generates mutants on method- and class-level [38, 51, 52]. Note that research on mutation testing for concurrent programs is also conducted [10], however, these are not utilized in our experiment. MuJava does not support mutation of concurrent code, thus faults has to be seeded by hand. This does not guarantee an unbiased seeding. Manually seeding of faults would also require a concurrency analysis of the code, which is beyond the scope of this experiment.

In addition, concurrent programs are not deterministic. This will likely cause different output from each and every execution even for pre-mutated test cases. This leaves us with more interpretation of the results because of the extra dimension added. At last, the degree of multithreading of the SUTs are not known. Given to SUTs, A and B, SUT A might be principally sequential compared to B.

MuJava is configurable to a certain extent. It allows selective creation of mutants, both traditional and class mutants. We chose to apply all possible mutation operators when mutation the software projects. Despite studies conducted on selective mutation [44,50], which effectively means that mutation operators are selected after to the *Pareto Principle*, which states that *..80 percent of the results stems from 20 percent of the work...* In our experiment, we did not want any mutation operator to escape or scrutiny, hence we chose to include all possible operators.

The final product from MuJava for this experiment is a directory structure containing mutants (mutated class files) which are organized by package and mutation operator. This concept is elaborated in Section 1.4.2. Unfortunately, not all classes were processed due to flaws in MuJava. Some were processed partially and some were omitted.

1.4.1 The JUnit framework

As we said in Section 1.4, criteria four, we support *JUnit* only. JUnit is a *framework for unit testing.* The developers of JUnit provide three rules that all unit test framework should adhere to [5, p. 8]:

Rule #1 Each unit test [case] should run independently of all other unit tests [cases].

Rule #2 The framework should detect and report errors test [case] by test [case].

Rule #3 It should be easy to define which unit test [cases] will run

The JUnit-book says: *..in order to for each unit test to be truly independent* [by rule #1]*, each should run in a different class loader instance.* This is the case as for every test case executed by the framework, a new JVM is started.

Rule #2 is also satisfied, as the smallest runnable entity for the framework is a test case. When this test case is executed, the results of all test methods encompassed are reported.

Rule #3 is trivially satisfied from our adherence to rule #2.

1.4.2 Assessment framework

We constructed a framework to handle execution of unit tests. This framework does not have a name and is interchangeably referred to as *the framework* or *our framework* henceforth. The premise for the framework is that it should process any valid input in a deterministic fashion and report errors that are produced during test case execution. The input to the framework is, on a high abstraction level, a *program*, *mutants* for the program and *test cases* for the program (Figure 1.2).

The framework should also provide the same environment for every execution, which is crucial to eliminate errors in the data sampling. From its design, the framework conforms to the rules stated by the JUnit developers.

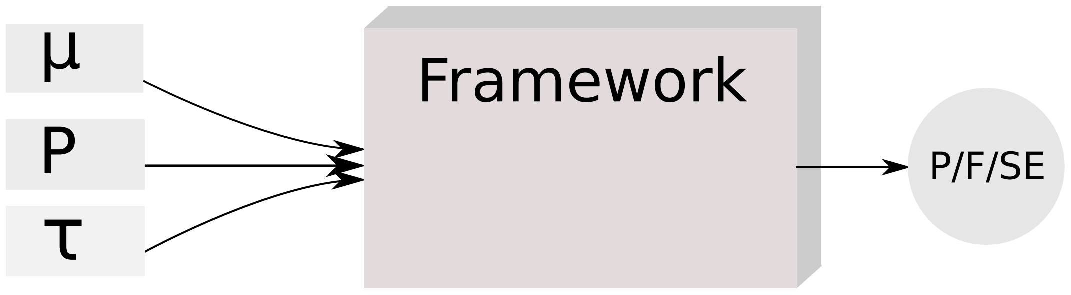


Figure 1.2: Abstract overview of the framework. Mutation operators, a program and a test suite is input, the output are results enumerated to pass (P), failure (F) and security exception (SE).

The framework is written in PERL and Java and requires that the SUTs are processed by MuJava in advance. It requires two different directory structures to operate; the byte code tree (the SUT) and the mutant tree (Figures 1.5 and 1.6 respectively). When a test case is executed, an a priori unknown number of files are utilized. These files may be *class-files*, *jar-archives* or other files related to the test case for proper execution. This is accomplished by utilizing a mechanism which is capable of (among other things) logging every access to the OS resources requested by the JVM. This mechanism is a sub class of *java.lang.SecurityManager* and is referred to as the *Custom Security Manager* (CSM). CSM can be visualized by a layer directly under the JRE (figure 1.3). Every request commissioned by the Java application is executed the JRE. Before it is passed to the OS, it is intercepted by the CSM, which is given the opportunity to stop the execution. See Section 2.3 for more details on CSM.

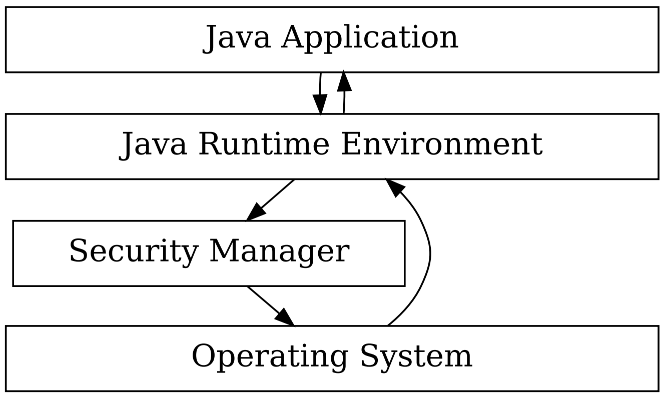


Figure 1.3: An abstract and simplified overview over the Java Runtime Environment with an active security manager. Note that a callback from the OS is returned directly to the JRE.

When the initial execution of the pre-mutated test case commence, the information about the set of *class files* that are accessed (or exercised) during the execution of a given test case is assembled and a data structure is created (Figure 1.4). Hopefully, the reader recognized this as a *hash-table* with the name of the test case as key and a list of the encompassed classes as its value. This structure provides the framework with knowledge in advance of which classes that will be implicated in the execution of a test case. This is important, because each of these classes needs to be replaced with mutated version for every execution of the test case. The hash table provides quick access to information about which classes to manipulate for any test case with a run-time of *O(1)* for every request. This ensures that every class and every mutant of the current test case is exercised.

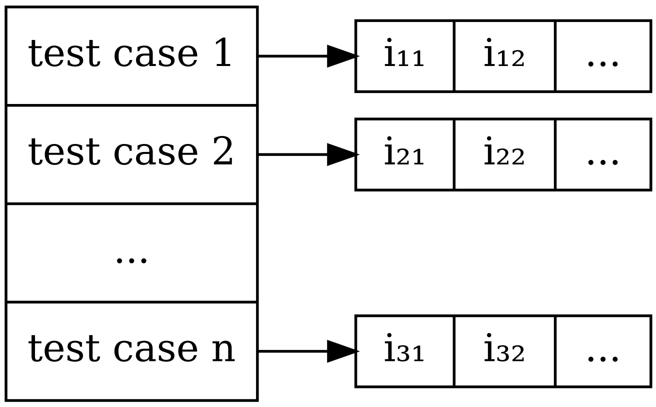


Figure 1.4: Diagram of the data structure which relates classes to their test cases. This is utilized when test cases are executed. The framework may easily obtain all implicated classes for any test case by providing the name of the test case.

For each test case in the test suite, one execution is performed by utilizing the pre-mutated version of the test case. The security manager is configured to allow any operation. An exhausting list of operations performed during the execution is recorded and two sets are derived from this information, i.e. the records of the data structure (Figure 1.4) and other files required to run, as described above. The second set of files, which consists of class files, jar-files and other vital files, is assembled and a *policy file* is created. The policy file and its context is described more detailed in chapter 2, Sections 2.1.1 and 2.1.2. For now, please regard this concept as a list over allowed operations. After the policy file is created, the CSM uses this for detecting access violations. For simplicity, we can say this is a *test oracle*. When policy files from each and all test cases are collected, the mutation assessment can begin.

The mutation assessment follows a pattern of execution similar to the process above. The difference is that the test case will undergo one additional execution for each mutant encompassed. The first execution is performed with the pre-mutated version of the test case, the rest with the different mutants. Basically, this means *n* mutants require *n + 1* executions. The latter *1* execution is performed with the pre-mutated test case. The remaining test cases will not commence unless this yields P.

The security manager is enforcing the permission policy by verifying each and every operation against the set of allowed operations described by the policy file. If, for instance, a file that is not specified in the policy file is accessed, we say that a *violation* has occurred. The policy file and the CSM serves as the *test oracle* and is the sole arbiter when decisions about security exceptions are made.

When a violation occurs, it is usually caused by a method encountering an abnormal condition that can not be handled by the method itself. The customary response to such a condition is to *throw an exception.* An exception that is thrown is an object with a type (just as any java object) which for simplicity can be described analogous to an *alarm.* (Not an alarm as in *POSIX*, but an everyday alarm, e.g. fire-alarm.) Just as a fire alarm requires an action or a way to *handle* it (which usually is well defined in any developed society), an exception in mature programs also have well defined handling mechanisms. In our experiment, a specific type of exception is thrown, namely the *java.security.AccessControlException* (ACE). This exceptions is not thrown by any method in the test case, but by the security manager. It follows the same pattern as above; the security manager intercepts an illegal operation and is signaling this to the JRE by throwing a security exception.

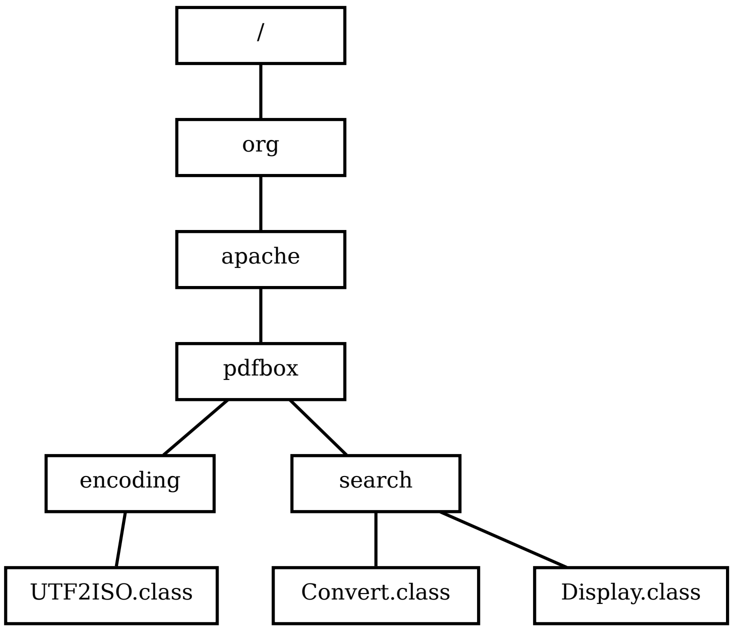


Figure 1.5: An excerpt from the byte code tree for PDFBox. Assume the tree classes depicted are encompassed by a test case. Fault seeding is done by replacing these class files prior to execution of the test case before execution.

Technically, this exception is inherited from *java.lang.SecurityException* which super class is the *java.lang.Exception.* Please see chapter 2 for more on access control exceptions.

The CSM is active during execution and enforces a pessimistic security policy, i.e. every operation *not* specified by the policy file is a security breach. When a test method employs objects that tries to access a resource that is not specified, the CSM will intercept this request and prohibit access. The very next step for the CSM (when running in a sequential environment) is to throw a security exception (an ACE in particular).

Formally, this decision process can be expressed as:

Let Σ be the *set of all resources accessed by the pre-mutated test case*
Let Σ^* be the *set of all resources accessed by the mutated test case* If $\Sigma^* \setminus \Sigma$ is not equal to $\emptyset$ then *throw new SecurityException*

An ACE is thrown iff $\Sigma^* \supset \Sigma$. Any operation omitted when a mutation operator is applied will therefore proceed unforeseen. There is no reason to believe that a set of operations for a pre-mutated test case is becoming hazardous if some of the operations are omitted, hence we do not address this issue.

Normally, an ACE is handled, but in our case it is not. The only thing that is of interest is that the ACE occurred. If this is the case for a test case, the test is terminated and the outcome is logged. Then the next test case is started.

When the process is complete, we can start to process the log files that are produced project-wise. The contents of these files are:

- name of the test case
- name of the mutant

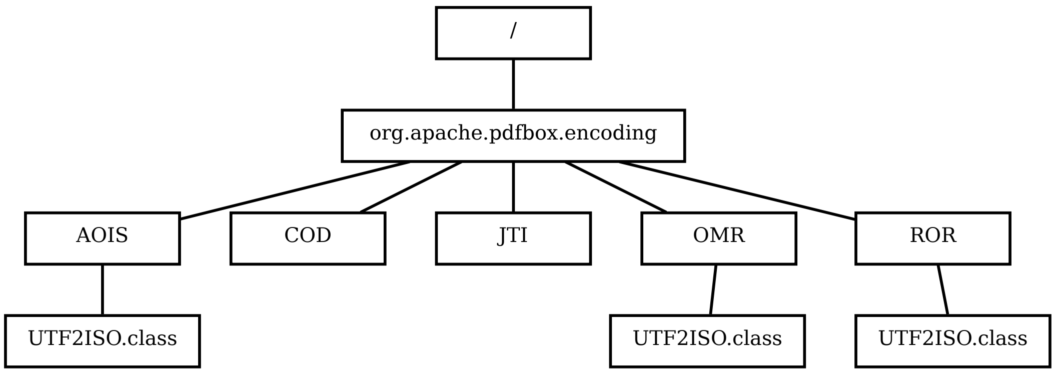


Figure 1.6: The mutant tree for PDFBox. All mutants are organized by full name of the original class. Mutants are organized under the mutation operator eligible to the original class, ensuring the correct class is being replaced by the correct mutant prior to test execution.

- name of the mutation operator
- description of the operation in case of a security exception, such as:
  - type of violation (file access, property access)
  - name of the target if applicable (object, file, property)
  - operation requested
- error traces are also possible

This information provides the basis of an extensive statistical analysis of the whole process. The error trace provides the opportunity to visit the source code to "see" what happened is also a possibility. On these foundations we will try discover trends for the mutation operators which will allow us to draw conclusions about the element of hazard attributed to a particular mutant operator. Eliminating such hazards will improve the method of mutation testing.

1.5 Thesis Structure

This thesis is structured as follows: Chapter 2 describes the Java policy model and provides some examples of security exceptions from the test case executions and a canonical example. Chapter 3 presents the results and an analysis. Chapter 4 provides discussion about the results and impact on the mutation testing process. Chapter 5 identifies threats to validity. Chapter 6 describes challenges that were met during development and execution. Chapter 7 concludes the thesis by summarizing the main results. Finally, an appendix is provided for the interested reader.

Chapter 2

Security Exception Sources

Java implements a security architecture providing the possibility to permit or prohibit operations. This can be done with high granularity, e.g. on file and socket level. This is governed by the *java.lang.SecurityManager* (SM), described introductory. When the SM is activated, it performs a pessimistic enforcement of the security w.r.t. accesses to resources. The SM intercepts every operation and it will only be permitted if a corresponding *java.security.Permission* object is found. These permission objects are either created from a policy file or programmatical as any general object during execution.

The SM contains a method *checkPermission(Permission perm)* which is utilized for every operation that is performed during execution. The parameter is a permission object enclosing information about which operation that is requested. By overriding this method in a subclass of the SM, we are able to deploy a security manager which implements business logic that directs this information to e.g. a *log file.* This subclass of the SM is referred to as the *Custom Security Manager* (CSM). We have already seen that the CSM is of high importance for accomplishment of our objective.

2.1 The Java Security Model

A logical diagram of the Java 2 Security Model (JSM) is depicted in figure 2.1. The fundamental component is the SM. The SM employs another mechanism, the *java.security.AccessController* (ACL). The ACL is able to control access with high granularity w.r.t. which *classes or code bases* that are granted access to different system resources.

It is in fact the ACL that is the governing mechanism regarding the prohibition of accesses. The employment of the ACL by the SM is transparent in the means that the ACL is being utilized to control accesses. For simplicity, we refer to the security component as the SM, even though the ACL is highly involved.

2.1.1 Permissions

A permission represents an *access to a system resource.* It typically has a name and a list of allowed actions. The name is often referred to as the *target name.* A permission that will permit read and write access to a file called *.shadow*[1] at the top level of a UNIX-like system may be specified (in code) like this:

```
myPermission = new java.io.FilePermission("/.shadow", "read, write");
```

[1]Note that the host's permissions is not overridden by the JVM. Most likely, access will be denied by the host.

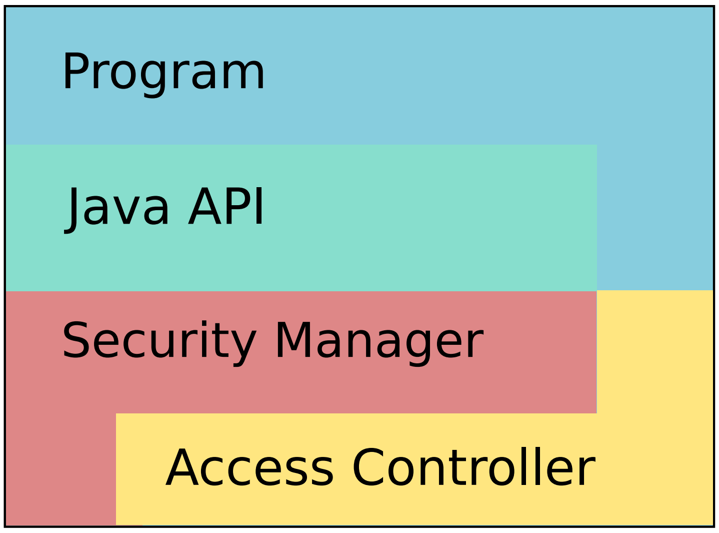


Figure 2.1: The Access Controller is the principal part of the Java Security Model. It is deployed by the Security Manager to control access with high granularity.

The target name in this case is the file"/.shadow". A target for a permission object is not necessarily a file. For instance, a *propertypermission* has a property as its target.

2.1.2 Policies

Instead of programmatically creating the required permissions, one may provide a *policy file*. When this is supplied to the JVM together with the SM parameter, permission objects are created implicitly. The policy file specification which is logical equivalent to the permission object in Section 2.1.1 (implicit creation of permission objects) follows:

```
grant {
    permission java.io.FilePermission "/.shadow", "read, write";
};
```

When the latter is absorbed by the JVM, this construct is logically equivalent to the statement in 2.1.1.

To generate policy files, we need execute a pre-mutated test case. The output of this execution is assembled and the data is structured identical to the permission above.

Other permissions that may be contained in a policy file are:

```
...
permission java.lang.RuntimePermission "createClassLoader";
permission java.util.PropertyPermission "java.system.class.loader", "read";
permission java.io.FilePermission "/tmp/-", "read, write, execute, delete";
...
```

The first allows the thread of execution to create a *classloader*-object. The classloader is a mechanism that simply loads a class' into the memory space of the JVM. The second grants the same thread to read *java.system.class.loader*, which would be prohibited of the entry is omitted. A qualified guess would be that the first permission is pointless upon this omission. The third permission is an interesting one. Note the hyphen after the directory specification; this allows the program to perform every known file system operation to the *tmp*-folder on the root of the file system and all files and folders residing directly below. Keep this in mind, as this peculiarity is central in the example in section 2.6.1.

2.2 Native Security Manager - java.lang.SecurityManager

This is the default and native SM [2, p. 877] which is activated either from the command line interface or programmatical. When active, SM intercepts all operations requested by the executing Java program and *checkPermission()* is called for each request. checkPermission() compares the operation with the its security policy (which is created either from file or direct creation of permission objects.) The SM is given the opportunity to stop an operation or let it complete. By default, when running Java applications, no security manager is active. This means effectively that every operation possible is allowed.

2.3 The Custom Security Manager

It is possible to subclass the native SM to create a custom built SM. In our case, this is crucial in order to create policy files automatically from execution of test cases. The native security manager will basically to the same job in context of restricting unauthorized operations. The difference is that with the CSM, the method *checkPermission()* is overridden and contains business logic to record the details of each an every operation requested and transpiring by the executed code. These details are logged to a file, *security.log*. In addition they are sent to standard out, which will allow the framework to capture output for immediate processing. These properties makes creation of policy files and the data structure (Figure 1.4) possible.

2.4 Automatic Creation of Policy Files

From the information in *security.log*, we create policy files for each test case. (Recall that the test case is the smallest runnable entity in our context, our framework does not support higher granularity such as executing single test methods.)

Listing 2.1 contains a fragment of the security log from the pre-mutated execution of the test case *org.apache.mina.statemachine.transition.MethodTransitionTest*.

Listing 2.1: Mina Security Record

```
<record>
  <date>2011-02-10T17:45:01</date>
  <millis>1297356301054</millis>
  <sequence>43</sequence>
  <logger>no.ronnyma.MySecurityManager.CustomSecurityManager</logger>
  <level>INFO</level>
  <class>no.ronnyma.MySecurityManager.CustomSecurityManager</class>
  <method>checkPermission</method>
  <thread>10</thread>
  <message>|class java.io.FilePermission|/home/ronnyma/Development/mscience/
      SUT/mina/test/org/apache/mina/statemachine/transition/
      MethodTransitionTest.class|read|</message>
</record>
```

In line 311, the *message* from the security manager conveys information about a permission, a file URI and an operation. The corresponding entry in a policy file created from this information and that would grant read access to *MethodTransitionTest.class* is:

```
grant {
...
    permission java.io.FilePermission "MethodTransitionTest.class", \
"read";
...
};
```

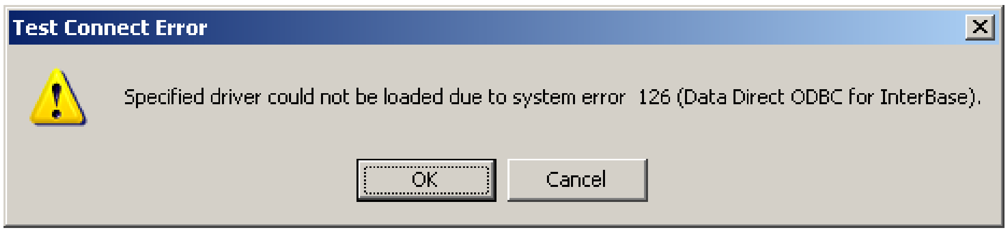
This operation is performed for every access to a resource when executing the pre-mutated test case. This ensures us that every resource that is accessed are logged and that reliable policy files can be created from this information.

2.5 Unstable Host Computer

We know that a Java program requires a JRE to execute and that the JRE requires a host computer (referred to as the *host*) with a compatible OS which in turn executes the JRE. A host comprises a great deal of libraries, executables, meta data which is required for proper operation. These entities are persisted as *datafiles* (referred to as just *files*) which exist in the file system.

When vital files are accidentally deleted from the host (or as a part of an assailed attack, just for the record), the host is prone to become unstable. Missing libraries, for instance would reduce the register of operations crucial for the host by removing required functions.

The consequences may be all from symptoms of malfunction to errors (Figure 2.2). A missing driver file could probably render a test case *failed*, which in the full extent will report a false negative, with the premise that the code which is tested by the test case is correctly written. A more severe consequence of a deleted file is *kernel panic* which is an *action taken by an operating system upon detecting an internal fatal error* [which may be caused by missing kernel modules] *from which it cannot safely recover* (Figure 2.3). This is equivalent to a *blue-screen* [69] on Windows Systems. This incident requires a restart of the host.

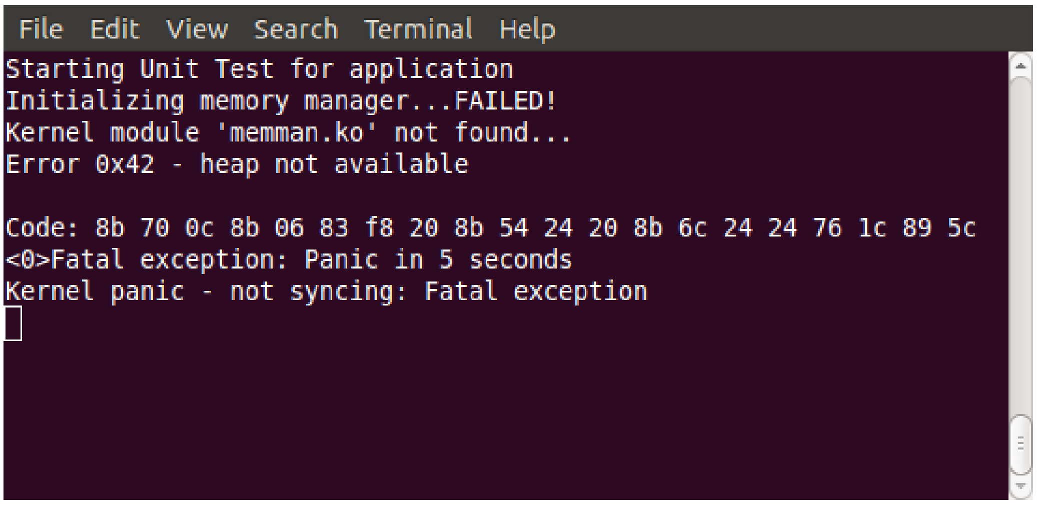

Test Connect Error

Specified driver could not be loaded due to system error 126 (Data Direct ODBC for InterBase).

Figure 2.2: An example of an error message on the Windows platform caused by a missing library. The file containing the library is for some reason missing.

In the context of automated testing, such incidents can be hazardous. There is no reason to believe that a successful test execution can be accomplished with errors such as the one in Figure 2.2 arised, let alone a kernel panic. Furthermore, unattended execution is of high importance. System errors as these are great antagonists to automated testing processes, be it testing processes in general.

To substantiate the understanding of the correlation of mutated code and exceptions we provide an analysis and a dry-run of the four permission violations causing the access control



```
Starting Unit Test for application
Initializing memory manager...FAILED!
Kernel module 'memman.ko' not found...
Error 0x42 - heap not available

Code: 8b 70 0c 8b 06 83 f8 20 8b 54 24 20 8b 6c 24 24 76 1c 89 5c
<0>Fatal exception: Panic in 5 seconds
Kernel panic - not syncing: Fatal exception
```

Figure 2.3: Kernel panic on a *NIX-system. To recover a kernel panic, a restart is required. In addition, the cause must be traced and any potential missing libraries needs to be restored.

exception (which is a direct subclass of security exception). The most important violation is of the java.io.FilePermission which we believe may cause the host to de-stabilize.

2.6 AccessControlException

From Oracles Java Documentation [42]: *This exception is thrown by the AccessController* [employed by the Security Manager] *to indicate that a requested access (to a critical system resource such as the file system or the network) is denied.*

When such a violation occurs from running the mutated test case, with the precondition that the pre-mutated version executes as expected, it is an indication of that the mutant caused that security exception. The environment of execution is set to be equal for each run (memory space, set-up and tear-down methods of JUnit with proper scaffolding), hence the only discrepancy is represented by the mutated class file executed by the test case. This suggests that the mutated code in the current execution space is the culprit.

2.6.1 FilePermission

There are four operations than can be performed on files and folders within the context of the JVM: read, write, delete and execute. These permissions are governed by the security manager and a policy specification. The security manager is given the opportunity to prevent completion of the requested operation.

To ease the comprehension of the consequences of a mutant causing FilePermission-violation, we have provided a canonical example of such a scenario (Listing 2.4). Please note that this example is trivial and constructed with the **sole purpose of demonstration.**

The cycle of execution starts by creating an instance of the class *Cleaner* with a location on the file system passed to its constructor, in this case *"tmp/work"*. This is a path relative to the root of the file system. Cleaner constructs the target path by concatenating the existing value of the field *"path"* with it self and the parameter from the constructor, i.e. the variable *"path"*. When this instantiation is done, the method *cleanDir* of Cleaner is called. Then the directory *"/tmp/work"* is traversed and all files and folder beneath are deleted recursively.

Listing 2.2: Canonical example

```
 1 import java.io.*;
 2 
 3 public class MalFile {
 4     public static void main(String[] args) {
 5         Cleaner cl = new Cleaner("tmp/work");
 6         cl.cleanDir();
 7     }
 8 }
 9 
10 class Cleaner {
11     public Cleaner(String Path) {
12         this.Path = this.Path + Path;
13     }
14 
15     public void cleanDir() {
16         File fullPath = new File("/" + this.Path);
17         recursiveDelete(fullPath);
18     }
19 
20     private void recursiveDelete(File dirPath) {
21         String[] ls = dirPath.list();
22 
23         for (int idx = 0; idx < ls.length; idx++) {
24             File file = new File(dirPath, ls[idx]);
25             if (file.isDirectory())
26                 recursiveDelete(file);
27             file.delete();
28         }
29     }
30 
31     private String Path = "/";
32 }
```

When we apply the mutation operator *JTI* [52], the code is modified.

Listing 2.3: Canonical example

```
11     public Cleaner(String Path) {
12         this.Path = this.Path + Path;
13     }
```

The mutation JTI is applied

$$(line\ 12)\ Path => this.Path \tag{2.1}$$

Listing 2.4: Canonical example

```
11     public Cleaner(String path) {
12         this.path = this.path + this.path;
13     }
```

The execution of this results in failure to a proper initialization if the field *"path"* which denotes the top level of the working directory. The field *"path"* will then assume the value "//" which effectively is the root of the file system. When *recursiveDelete* is called, Java will sift through the whole file system, diving into folder recursively, deleting everything it encounters[2]

This is an extreme example, regarding both the semantics of the code and the outcome of the execution. Nevertheless, it does not suggest that production code does not possess the same arcane properties or that other code structures will not encounter similar changes in semantics after a mutant is applied. More specific examples from test subjects are provided below.

The first example is from BCEL. Given code Listing 2.5, it specifies a method that returns an object of type *ClassFile* based on the existence of a file represented by the field *file*. According to its documentation: *"Responsible for loading (class) files from the CLASSPATH. Inspired by sun.tools.ClassPath."*

After inspecting the code, we see that *file* is initialized with an URI to a class file which is subject to loading into the JVM. Line 435 replaces the *dots* with *slashes* in *name* which indicates a translation from a full qualified class name to a folder structure containing the physical class file. (A *dry-run* session confirmed this.)

Listing 2.5: BCEL ClassPath getClassFile

```
423 ClassFile getClassFile( String name, String suffix ) throws IOException {
424     final File file = new File(dir + File.separatorChar
425             + name.replace('.', File.separatorChar) + suffix);
426     return file.exists() ? new ClassFile() {
427 
428         public InputStream getInputStream() throws IOException {
429             return new FileInputStream(file);
430         }
431 
432 
433         public String getPath() {
434             try {
435                 return file.getCanonicalPath();
436             } catch (IOException e) {
437                 return null;
438             }
439         }
440 
441 
442         public long getTime() {
443             return file.lastModified();
444         }
445 
446 
447         public long getSize() {
448             return file.length();
449         }
450 
451 
```

[2]The outcome depends on permissions and the account the program is executed on behalf of.

```
452         public String getBase() {
453             return dir;
454         }
455     } : null;
```

After LOI was applied, a test execution reported this:

```
java.security.AccessControlException
(testFieldAnnotationEntrysReadWrite)
( access denied
(java.io.FilePermission target/testdata/./-47AnnotatedFields.class
read))
```

Clearly, *-47AnnotatedFields.class* was not accessed by the original test case, i.e. does not exist in the **set of permitted files**. (It might not even exist physically.) To get a comprehension of what caused this incident, we need to examine the source code of the class causing it. The relevant source code segment, original version and mutated version, is listed below (Listing 2.6 and 2.7). As we can see from these excerpts, an abstract representation of a file is created from the variable *dir* concatenated with the file separator character defined by the system, the fields *name* and *suffix*. A closer look reveals that a minus sign is put in front of the *File.separatorChar* causing the filename to be different from the filename specified in the security policy file, hence a security exception is thrown.

Listing 2.6: BCEL ClassPath

```
423 ClassFile getClassFile( String name, String suffix ) throws IOException {
424             final File file = new File(dir + File.separatorChar
425                     + name.replace('.', File.separatorChar) + suffix);
426             return file.exists() ? new ClassFile() {
427 
428                 public InputStream getInputStream() throws IOException {
429                     return new FileInputStream(file);
430                 }
```

The mutation LOI is applied:

$$(line\ 424)\ File.separatorChar => -File.separatorChar \tag{2.2}$$

Listing 2.7: BCEL ClassPath LOI

```
423 ClassFile getClassFile( String name, String suffix ) throws IOException {
424     final File file = new File(dir + -File.separatorChar
425                  + name.replace('.', File.separatorChar) + suffix);
426     return file.exists() ? new ClassFile() {
427                 public InputStream getInputStream() throws IOException {
428                 return new FileInputStream(file);
429              }
```

This mutation operator caused an *AccessControlException*, hence it is a strong mutation [27, 72]. The *null* returned from *getFilePath()* will propagate through the cycle of execution when the ternary condition construct in line 426, Listing 2.5 is fed with False from file.exists().

2.6.2 ManagementPermission

This permission has two target names; *control* and *monitor*. The latter were reported when runningVmPipeSessionCrossCommunicationTest in Apache Mina.

The *monitor* permission allows the ability to retrieve run time information about the JVM such as thread stack trace, a list of all loaded class names and input arguments to the JVM. The risks of allowing this permission is that malicious code can monitor run time information and uncover vulnerabilities.

From the Javadoc for org.apache.mina.transport.vmpipe (the package of VmPipe) we learn that "In-VM pipe support which removes the overhead of local loopback communication." VmPipe has a field of type *IoServiceListenerSupport* which is *a helper class which provides addition and removal of IoServiceListeners and firing events.*

Listing 2.8 shows the relevant code segment.

Listing 2.8: Mina VmPipe Excerpt

```
25 class VmPipe {
26
27     private final VmPipeAcceptor acceptor;
28
29     private final VmPipeAddress address;
30
31     private final IoHandler handler;
32
33     private final IoServiceListenerSupport listeners;
34
35     VmPipe(VmPipeAcceptor acceptor, VmPipeAddress address,
36             IoHandler handler, IoServiceListenerSupport listeners) {
37         this.acceptor = acceptor;
38         this.address = address;
39         this.handler = handler;
40         this.listeners = listeners;
41     }
42 ...
43 }
```

The violation that was reported:

```
access denied (java.lang.management.ManagementPermission monitor))
```

In listings 2.9 and 2.10, line 40 the delta is easily seen..

Listing 2.9: Mina VmPipe

```
35     VmPipe(VmPipeAcceptor acceptor, VmPipeAddress address,
36             IoHandler handler, IoServiceListenerSupport listeners) {
37         this.acceptor = acceptor;
38         this.address = address;
39         this.handler = handler;
40         this.listeners = listeners;
41     }
```

The mutation JTI is applied:

$$(line\ 40)\ listeners => this.listeners \tag{2.3}$$

Listing 2.10: Mina VmPipe JTI

```
35    VmPipe(VmPipeAcceptor acceptor, VmPipeAddress address,
36            IoHandler handler, IoServiceListenerSupport listeners) {
37        this.acceptor = acceptor;
38        this.address = address;
39        this.handler = handler;
40        this.listeners = this.listeners;
41    }
```

In our case, the application of the JTI-mutant impedes the initialization of the class field *listeners*. Its value was intended to be passed from the method parameter with type *IoServiceListenerSupport* and name *listeners*. However, the insertion of *this* transforms line 40 to an idempotent clause, hence no value is modified. Apparently, this propagated the whole call stack, thus being classified as a strong mutation.

2.6.3 RuntimePermission

RuntimePermission is a cousin of FilePermission, i.e. both are direct subclasses of *java.security.Permission*. RuntimePermission violations occur when a running program tries to access a system resource for which no access is granted. In our experimental case, the Security Manager is enforcing the security policy when test cases are executed, thus capturing this exception and reporting as an AccessControlException.

The Security Manager yields this message:

```
java.security.AccessControlException (test1)
( access denied
(java.lang.RuntimePermission accessDeclaredMembers))
```

When inspecting the original source code and the mutant, listings 2.11 and 2.12 respectively, we discover that a discrepancy leads to different properties are collected.

Listing 2.11: Log4J RendererMap

```
165 ObjectRenderer r = (ObjectRenderer) map.get(c);
166 if(r != null) {
167   return r;
168 } else {
169   Class[] ia = c.getInterfaces();
170   for(int i = 0; i < ia.length; i++) {
171   r = searchInterfaces(ia[i]);
172   if(r != null)
173     return r;
174   }
175 }
176 return null;
```

The mutation EAM is applied:

$$(line\ 169)\ c.getInterfaces() => c.getDeclaredClasses() \tag{2.4}$$

Listing 2.12: Log4J RendererMap EAM

```java
ObjectRenderer r = (ObjectRenderer) map.get(c);
if(r != null) {
  return r;
} else {
  Class[] ia = c.getDeclaredClasses();
  for(int i = 0; i < ia.length; i++) {
  r = searchInterfaces(ia[i]);
  if(r != null)
    return r;
  }
}
return null;
```

According to the Java Documentation the former accessor method *"determines the interfaces implemented by the class or interface represented by this object."*. The latter *"returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object (..)"* The policy file for the original code does not allow this to happen, hence an ACE is thrown.

2.6.4 SocketPermission

A socket is used by a Java program to connect to other hosts (or itself via the network). This permission is almost self explanatory, but a synopsis is that a SocketPermission consists of a host specification and a set of actions specifying ways to connect to that host, e.g. port number, protocol and $\{accept|connect|listen|resolve\}$.

In this case, the PCI operator emulates a programming error by inserting an incorrect type cast operator which casts from int to org.apache.mina.transport.socket.apr.AprSocketConnector.

Listing 2.13: Mina IOServiceStatistics

```java
256 private void resetThroughput() {
257     if (service.getManagedSessionCount() == 0) {
258         readBytesThroughput = 0;
259         writtenBytesThroughput = 0;
260         readMessagesThroughput = 0;
261         writtenMessagesThroughput = 0;
262     }
263 }
```

$$(line\ 257)\ service => ((AprSocketConnector)service) \tag{2.5}$$

Listing 2.14: Mina IOServiceStatistics PCI

```java
256 private void resetThroughput() {
257     if ((AprSocketConnector)service.getManagedSessionCount() == 0) {
258         readBytesThroughput = 0;
259         writtenBytesThroughput = 0;
260         readMessagesThroughput = 0;
261         writtenMessagesThroughput = 0;
```

```
    }
}
```

To find the root-cause of this violation, a thorough analysis of the source code of Apache Mina is required, which is beyond our scope. The most important factor is that the exception is manifest, which means that a violation occurred.

2.7 FilePermission violation

The most abundant of all ACEs that occurs in this study is caused by accesses to files by mutated test cases that were not accessed by the pre-mutated test cases, i.e. insufficient file permissions. This exception is an indication that a file-level violation has occurred. This incident is suspected to cause the most crash-prone scenarios, especially when the operation is *write* or *delete*. To elaborate the reader's comprehension of this violation, more examples are provided.

2.7.1 Apache Scout - EAM

This source of exception is somewhat peculiar. After visiting the source code of *java.lang.Throwable*, it is clear that the pre-mutated version and the mutant should return the same field. In the inheritance chain for this class,i.e. *org.apache.ws.scout.registry.RegistryException* these methods are never overridden. Anyway, this mutant is reported to cause a file access violation, thus it is included for demonstrative purposes.

Listing 2.15: Scout RegistryException:RegistryExeption

```
/**
 * Constructs a RegistryException instance.
 * @param ex the original exception
 */
RegistryException(String fCode,int errno,String msg)
{
  super(buildMessage(errno,msg));

  String errCode = lookupErrCode(errno);

  if (fCode != null) {
    setFaultCode(fCode);
  }

  setFaultString(getMessage());

  Result r = this.objectFactory.createResult();
  ErrInfo ei = this.objectFactory.createErrInfo();

  if (errCode != null) {
    ei.setErrCode(errCode);
  }

    ei.setValue(getMessage());
    r.setErrno(errno);

170
171     if (ei != null) {
172       r.setErrInfo(ei);
173     }
174
175     addResult(r);
176   }
```

Listing 2.16: Scout RegistryException

```
163     if (errCode != null) {
164       ei.setErrCode(errCode);
165     }
166
167       ei.setValue(getMessage());
168       r.setErrno(errno);
```

$$(line\ 167)\ getMessage() => getLocalizedMessage() \tag{2.6}$$

Listing 2.17: Scout RegistryException

```
163     if (errCode != null) {
164       ei.setErrCode(errCode);
165     }
166
167       ei.setValue( getLocalizedMessage() );
168       r.setErrno(errno);
```

2.7.2 PDFBox - AOIS

This violation occurs for the following reasons: Consider listing 2.15, line 169: retval, which later on is used as a filename is initialized. *getBytes()* returns a byte array containing a filename, *start* denotes the offset in the byte array (to start reading from), *data.length-start* is the amount of bytes to read and *encoding* specifies the encoding in which the string should be.

The delta applied by the AOIS (2.7) increases the integer *start*, thus the first byte is not included in the String-constructor.

This deviation propagates when the method returns, as *retval* is used as a filename. The value contained in *retval* is used as a file name and is not specified in the policy file, hence an access control exception occurs.

Listing 2.18: PDFBox - COSString:getString

```
144 public String getString()
145 {
146     if (this.str != null)
147     {
148         return this.str;
149     }
150     String retval;
151     String encoding = "ISO-8859-1";
152     byte[] data = getBytes();
153     int start = 0;
154     if( data.length > 2 )
155     {
156         if( data[0] == (byte)0xFF && data[1] == (byte)0xFE )
157         {
158             encoding = "UTF-16LE";
159             start=2;
160         }
161         else if( data[0] == (byte)0xFE && data[1] == (byte)0xFF )
162         {
163             encoding = "UTF-16BE";
164             start=2;
165         }
166     }
167     try
168     {
169         retval = new String( getBytes(), start, data.length-start, encoding );
170     }
171     catch( UnsupportedEncodingException e )
172     {
173         //should never happen
174         e.printStackTrace();
175         retval = new String( getBytes() );
176     }
177     this.str = retval;
178     return retval;
179 }
```

Listing 2.19: PDFBox - COSString:getString

```
263     try
264     {
265         retval = new String( getBytes(), start, data.length-start, encoding );
266     }
```

$$(line\ 265)\ start => ++start \tag{2.7}$$

Listing 2.20: PDFBox - COSString:getString

```
263     try
264     {
265         retval = new String( getBytes(), ++start, data.length-start, encoding )
                ;
266     }
```

2.7.3 PDFBox - JSI

This is an interesting case. The only delta is that a object field is modified to a class field, hence the static modifier keyword. A dependency analysis reveals that the class *COSString* is intricately involved in the class hierarchy of the test case that threw a security exception, i.e. depended on by quite a few of these classes. As we can see from listing 2.21, no *this*-keyword is used when on the variable modified by the mutation, hence the compiler will face no problems regarding if *out* is a class field or an object field. Alas, the semantics are being changed, and the adept Java programmer will understand that *out* is shared by all created instances of *COSString*. Having it mutable will not be an advantage either since a modification to this will be reflected in all instances of this class.

The exception message conveys that an access to an unexpected file is attempted.

```
java.security.AccessControlException (testExtract) (
access denied (
java.io.FilePermission org/apache/pdfbox/resources/cmap/Microsoft \
 Word - Document1-Microsoft Word - Document1-UCS2 read))
```

Listing 2.21: PDFBox - COSString:out

```java
    public COSString( String value )
    {
        try
        {
            boolean unicode16 = false;
            char[] chars = value.toCharArray();
            int length = chars.length;
            for( int i=0; i<length; i++ )
            {
                if( chars[i] > 255 )
                {
                    unicode16 = true;
                    break;
                }
            }
            if( unicode16 )
            {
                byte[] data = value.getBytes( "UTF-16BE" );
                out = new ByteArrayOutputStream( data.length +2);
                out.write( 0xFE );
                out.write( 0xFF );
                out.write( data );
            }
            else
            {
                byte[] data = value.getBytes("ISO-8859-1");
                out = new ByteArrayOutputStream( data.length );
                out.write( data );
            }
        }
        catch (IOException ignore)
        {
            ignore.printStackTrace();
            //should never happen
        }
    }
```

Here are two listing of the same code segment after applying 2.8. As we can see, *out* is utilized extensively by in 2.21.

Listing 2.22: PDFBox - COSString:out

```
78     private ByteArrayOutputStream out = null;
79     private String str = null;
```

$$(line\ 78)\ static\ is\ inserted \tag{2.8}$$

Listing 2.23: PDFBox - COSString:out JSI

```
78     private static ByteArrayOutputStream out = null;
79     private String str = null;
```

2.7.4 Maven3 - JTI

ReactorContext has-a ProjectIndex which is responsible for the book keeping of projects. *ReactorContext.getProjectIndex()* will return the ProjectIndex object. After 2.9 is applied, normal operation is abrupted. The adept java developer will understand that *null* is returned.

Listing 2.24: Maven ReactorContext

```
public class ReactorContext
{
    private final MavenExecutionResult result;

    private final ProjectIndex projectIndex;

    private final ClassLoader originalContextClassLoader;

    private final ReactorBuildStatus reactorBuildStatus;


    public ReactorContext( MavenExecutionResult result, ProjectIndex projectIndex,
                           ClassLoader originalContextClassLoader, ReactorBuildStatus reactorBuildStatus )
    {
        this.result = result;
        this.projectIndex = projectIndex;
        this.originalContextClassLoader = originalContextClassLoader;
        this.reactorBuildStatus = reactorBuildStatus;
    }

    public ReactorBuildStatus getReactorBuildStatus()
    {
        return reactorBuildStatus;
```

```java
    }

    public MavenExecutionResult getResult()
    {
        return result;
    }

    public ProjectIndex getProjectIndex()
    {
        return projectIndex;
    }

    public ClassLoader getOriginalContextClassLoader()
    {
        return originalContextClassLoader;
    }

}
```

In the original code segment, *this.projectIndex* is initialized in the constructor.

Listing 2.25: Maven ReactorContext

```java
    public ReactorContext( MavenExecutionResult result, ProjectIndex projectIndex,
                           ClassLoader originalContextClassLoader, ReactorBuildStatus reactorBuildStatus )
    {
        this.result = result;
        this.projectIndex = projectIndex;
        this.originalContextClassLoader = originalContextClassLoader;
        this.reactorBuildStatus = reactorBuildStatus;
    }
```

$$(line\ 24)\ projectIndex => this.projectIndex \tag{2.9}$$

This modification causes *this.projectIndex* to have a object fields default value.

Listing 2.26: Maven ReactorContext

```java
    public ReactorContext( MavenExecutionResult result, ProjectIndex projectIndex,
                           ClassLoader originalContextClassLoader, ReactorBuildStatus reactorBuildStatus )
    {
        this.result = result;
        this.projectIndex = this.projectIndex;
        this.originalContextClassLoader = originalContextClassLoader;
        this.reactorBuildStatus = reactorBuildStatus;
    }
```

All real examples are extracted from the test subjects (the projects) and are selected by random. When a test case fails, the cause is attributed to one mutant only since we never apply more than one mutation operator per execution.

The next step in our experiment is to see if some of the mutation operators are more prone to make the subject class to cause security exception than other mutation operators. To discover a trend, several parameters need to be accounted for, e.g. the proneness a class possess *per se* for making a test case fail, i.e. if a class is inclined to cause the test case to fail even when no mutation operator is applied.

Chapter 3

Results and Analysis

When all the results from the executions of the mutants are collected, a thorough analysis is required. We wish to see if there is a trend for the mutation operators, i.e. if a **statistical** relationship between a security exception and the mutant exists. Note that the word statistical is emphasized to stress that we do not expect to find a *deterministic* relationship.

The quantitative metrics such as *#classes* (number of classes), *#methods* (number of methods), *LOC* (line of code), *#Classes (T)* (number of classes encompassed by the test case), *LOC (T)* (lines of test code) were collected with Understand$^{\text{TM}}$for Java. Code coverage (denoted by just *Cover*) were measured by Code Cover for Maven from Atlassian$^{\text{TM}}$(full functional trial version.)

The percentage of outcome, i.e. *passed* (P), *failed* (F) and *security exceptions* (SE) produced by the test cases was gathered from the log files with bash-scripts and the UNIX-commands *sed*, *awk*, *sort* and *uniq*.

Violations are: *FP*, *RP*, *SP* and *MPM* which are *FilePermission*, *RuntimePermission*, *SocketPermission* and *ManagementPermission* respectively.

The *test code coverage* (denoted just *Cover*) was obtained from *method coverage*, *statement coverage* (aka. *basic block coverage*) and *conditional coverage* aggregated (see table 3.2). No whitespace or comments are included in the metrics.

Box plots, histograms and Kruskal-Wallis are generated with R. Other plots are generated with Matlab.

3.1 Test Subjects and Characteristics

From the set of all Maven enabled projects, the set of test subjects (projects) were selected from the specification in Section 1.4. Their application areas implies that they necessarily possess different qualities regarding operations performed during execution.

Take for instance BCEL *(...intended to give users a convenient possibility to analyze, create, and manipulate ... Java class files)*, Maven 3 *(tool for building and managing any Java-based projects*), Log4J *(...a tool to help the programmer output log statements to a variety of output targets)*, Apache Scout (an implementation of *Java API for XML Registries [JAXR])* and PDFBox *(...allows creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents).* The description provided by the developers suggests that file accesses happens frequently, hence we would expect a higher rate of file access violations compared to *Apache Commons Math* when executed within our framework. From the home page of the latter: *...library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language or Commons Lang.*

By code inspection, we discovered that *Apache Commons Math* consists of quite a few computationally intensive routines, but very few file accesses. A preliminary expectation is a lot of failed test cases, due to errors in output compared to the expected output given by the oracle. These errors are expected to be caused by mutation operators that makes modifications to arithmetic operators. A quick review of the code revealed nested loops enclosing logic containing basic numerical operators for *addition* and *subtractions* among others.

We therefore expect different results regarding file access violations when different projects are evaluated in the context of mutation testing.

We will refer to our test subjects quite a few times in this chapter, thus we choose to identify them by an index (table 3.1). For instance, when we say project 1, we mean BCEL.

Table 3.1: Henceforth, projects are identified by an integer.

Project	Index
BCEL	1
Apache Cactus	2
Maven 3	3
Squirrel SQL	4
Struts	5
Log4J	6
Apache Commons Math	7
Apache MINA	8
PDFBox	9
Apache Scout	10

3.2 Test Subject Metrics and Statistics

For starters, we provide some quantitative metrics of the test subjects. As we can read from table 3.2, the quantitative dimensions differ quite a bit, e.g. projects 4 and 5: The latter outnumbers the first regarding *number of classes* in approximately a magnitude of six. Measured in LOC, the same parameter is approximately four. A further review indicates a difference in nature for all the projects. On these grounds, we justify that our sample possesses a diversity w.r.t. to measured quantities.

The code coverage ratio ranges from 9.50 to 88.30. The low ratio may suggest badly written test cases, which again will affect the quality of the outcome. This is because low quality test cases detect less malicious results than its cousins with higher quality. Mutants that escape the test execution will definitely not throw a security exception.

Table 3.2 shows different metrics for the projects, among the *test code coverage.* Notice that no data for projects 2 and 5 exists, because we were unable to run a coverage assessment for these projects. Nonetheless, we will try to alleviate this by investigating the metrics for the other projects. This will allow us to make suggestions about the coverage for projects 2 and 5.

Table 3.2: Quantitative metrics from the executions of mutants.

Project	#Classes	#Methods	LOC	#Classes (T)	LOC (T)	#Muts.	Kill Muts.
1	468	3735	34393	21	1916	55085	3157
2	417	2103	28279	71	5958	1445	302
3	1063	6433	92086	258	15695	13350	5176
4	4475	17933	235891	612	35041	237016	8370
5	719	6322	65320	672	6360	11338	997
6	531	2473	34461	134	9114	99097	21697
7	1091	4679	88147	486	43829	423009	55820
8	838	4686	46143	171	17790	66447	28905
9	528	4282	52787	28	1939	47346	16646
10	189	1464	18630	34	3852	2919	1680
Average	1032	5411	69614	249	14149	90705	5934

Project	% Cover	% Pass	% Fail	% SE	% FP	% RP	% SP	% MPM
1	20.00	94.27	5.72	.01	100	0	0	0
2	N/A	79.10	19.17	1.73	100	0	0	0
3	31.40	61.29	37.39	1.39	100	0	0	0
4	15.10	96.47	3.53	0	0	0	0	0
5	N/A	91.21	8.79	0	0	0	0	0
6	36.50	78.11	21.89	.01	0	100	0	0
7	88.30	86.79	13.21	0	0	0	0	0
8	40.50	56.49	38.62	4.88	0	0	99.99	.01
9	9.50	64.84	35.11	.05	0	0	0	0
10	38.00	42.45	57.52	.03	100	0	0	0
Average	28	75.10	24.1	.81	40	10	10	0

Table 3.3: This table shows that there are only small discrepancies with respect to the different coverage measurement models. Unfortunately, Cactus and Struts did not execute after the instrumentation by *Cover*.

Project	m-cover	s-cover	c-cover	(Cover)
1	24.40	21.00	14.10	20.00
2	N/A	N/A	N/A	N/A
3	32.30	32.30	8.60	31.40
4	21.20	14.70	11.80	15.10
5	N/A	N/A	N/A	N/A
6	38.00	36.40	35.90	36.50
7	84.10	89.80	86.40	88.30
8	41.60	42.10	35.00	40.50
9	9.10	9.80	8.90	9.50
10	44.90	38.80	30.80	38.00

3.2.1 Preliminary Data Analysis - Code Coverage for projects 2 and 5

From table 3.2 some questions can be answered, especially regarding (2 and 5) where coverage data is missing. We will try to make a suggestion about the code coverage by investigating characteristics of the other projects. Test code coverage is correlated to the robustness of a test case, hence it is important when we are discussing threats to validity later.

Goodness of Fit w.r.t. Mutant Kill Ratio

To alleviate the impact caused by missing coverage data for projects 2 and 5, we wish to find the correlation between code coverage and the mutant kill ratio for all projects (except from projects 2 and 5.) On this basis, we can make a suggestion concerning the code coverage for the projects with omitted coverage information. This is important for the validity of the data.

Now, since we already know the observed data (code coverage) and the expected data (mutant kill ratio), we want to employ Pearson's chi-square test to calculate χ^2 and then suggest whether projects 2 and 5 possess a decent test code coverage.

We state the null-hypothesis:

> $\mathbb{H}_0$: The distribution of killed mutants follows the code coverage distribution.

The alternate hypothesis:

> $\mathbb{H}_a$: The distribution of killed mutants does not follow the code coverage distribution.

Let O_i, {i $\in$ [1,10] $\notin$ 2,5} be the coverage ratios for test subject *i*. Let E_i, {i $\in$ [1,10] $\notin$ 2,5} be the relative mutant kill ratio for test subject *i*. Since we are operating with 8 values, the *degrees of freedom* (df) is 7. Let α be .05 denoting a confidence interval of 95 percent, we calculate the chi-square by employing *Pearson's chi-square test.*

The chi-square is then: $\chi^2 = \frac{(O_i - E_i)^2}{E_i} = 2664.2800$

From the chi-square table (Table A.4), the value for a 95 percent confidence interval lies below 14.07 having 7 degrees of freedom. Our χ^2 if definitely in the rejection region, hence $\mathbb{H}_0$ is rejected.

In this basis, we cannot make a suggestion of code coverage ratio for project 2 and 5.

Test Code Coverage vs Detected Failures

We still does not have any code coverage data for projects 2 and 5, hence we move on. Studies suggests that *test coverage* is highly correlated with *test suite effectiveness* [7, 28, 45]. This may aid us to make assumptions of the code coverage for projects 2 and 5.

This time we will investigate in which degree this applies to our test subjects as well. We start by plotting the coverage and the detected failures. Failures in this sense comprise both *failed test cases* and *security exceptions.*

We start by stating the null-hypothesis:

> $\mathbb{H}_0$ = Test code coverage is correlated with detected failures.

On the contrary, an alternate hypothesis is:

> $\mathbb{H}_a$ = Test code coverage is not correlated with detected failures[1]

According to Figure 3.1, it seems like the distribution is random. One would expect the dotted line to lie strictly below the solid.

Following the same procedure as Section 3.2.1, we calculate the χ^2 to be 537.4565. With *df* = 7 (projects 2 and 5 are omitted) and α = .05, the critical value is still 14.07, hence χ^2 is in the rejection interval. This approves $\mathbb{H}_a$, and $\mathbb{H}_0$ is rejected.

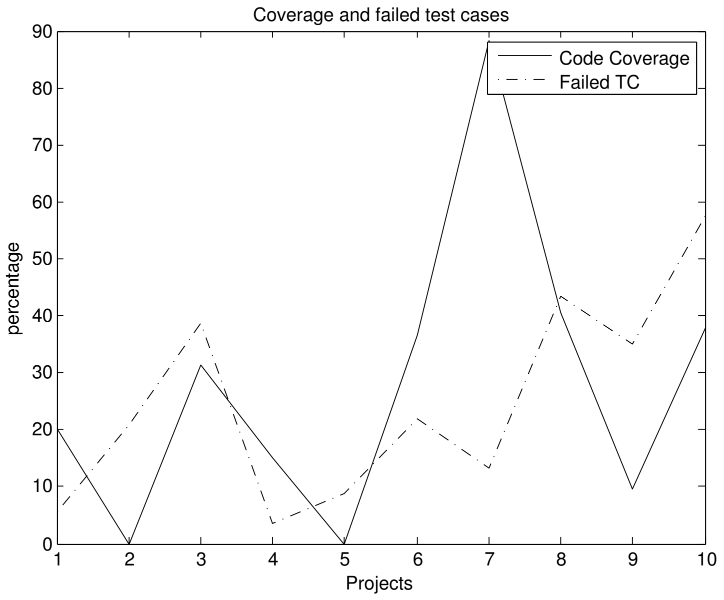


Figure 3.1: Test Code Coverage with failed tests, with a χ^2 = 537.4565. Note that projects 2 and 5 are omitted from the calculations.

Nonetheless, the studies [7, 28, 45] allows us to make an important assumption regarding projects 2 and 5, where no coverage data is reported. The fault-detection ratio (failures + mutants killed) is approx. 21 and 9 percent respectively. Considering the average coverage, which is 28 percent, this suggests that the coverage is average for 2 and substandard for 5. The latter assumption is based on [13, 71]. We choose to include these in the experiment, due to the average low coverage among the projects based on this assumption.

3.3 Research Questions

When running unit tests in the context of this experiment, the three possible outcomes are pass (P), fail (F) and security exception (SE). Recall that we smallest unit of execution is *one* test case, which may contain an arbitrary number of test methods. Each of these test methods are completely disjoint to the other test methods w.r.t dependency among them. This is synonymous with the claim that for two test methods t_1 *and* t_2, the operation of t_2 is not in any mean dependent on the operation of t_1. If, for instance t_1 initializes a variable which value functions as a *precondition* for t_2, then the two test methods are conjoined. This is not according to best practice.

[1]Failures in this sense comprise both *failed test cases* and *security exceptions.*

Let τ be a test case encompassing $n > 1$ test cases, $n \in \mathbb{N}$. Let P be the outcome iff all tests in τ passes. If one or more of the test cases yield a SE, the outcome is SE. If no SE is yielded and at least one of the test case yields an F, the outcome is F. See Figure 3.2. If $n = 1$, the result evaluation is trivial.

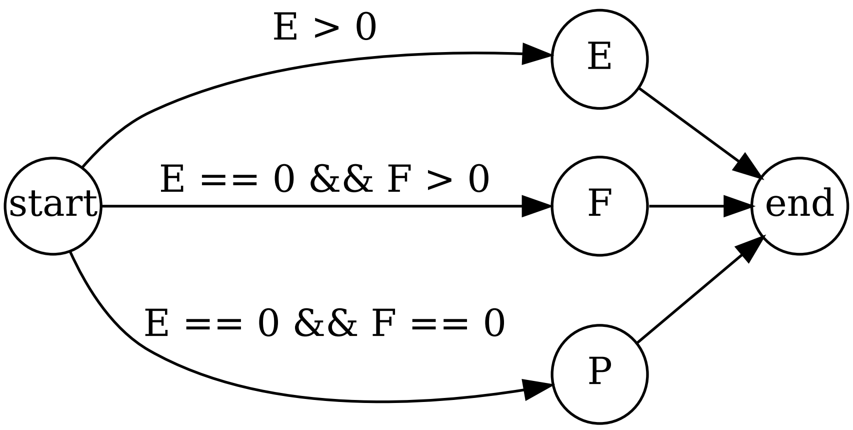


Figure 3.2: State Machine Diagram of the result evaluation process for mutants. Information about the outcome from the test cases in the log files is used as input. This is typically a list of results for a test case containing each distinctive result from the test methods of the test case. The output is a scalar.

After all projects were evaluated, we possessed information about the *test case* that were executed, the *class* that were mutated in this execution context and which *mutation operator* that spawned the mutant. This provides us with valuable information.

We will use the results from this process to answer the research questions below.

RQ1
What are the differences in proportions regarding P/F/SE among the different projects?

RQ2
What are the proportional variations in P/F/SE among the different mutated classes?

SRQ2.1 Are there properties to the each of the classes that makes it prone to yield a specific outcome?

RQ3
What are the proportional variations in P/F/SE among the different mutation operators?

SRQ3.1 Are there properties for some mutation operators that makes the classed inclined to yield a specific outcome?

3.3.1 RQ1

What are the differences in proportions regarding P/F/SE among the different projects?

Most of the mutants that were exercised during execution of the test cases yield P. This suggests that test cases are inept of detecting faults, which reflects the low test code coverage. From Figure 3.3, the proportions are easily identified. Overall, there is just a few security exceptions. Some of the projects did not yield security exceptions at all, thus deteriorating our investigational basis.

The deviant is project 10, its ratio of failed test cases are higher than the other projects. Initially, we were inclined to believe that the test cases possessed higher quality than for the other projects. Alas, these results are due to projects 10 sparse number of few classes that were exercised during the test. A few failures will have a visible impact on the outcome, which is evident for project 10.

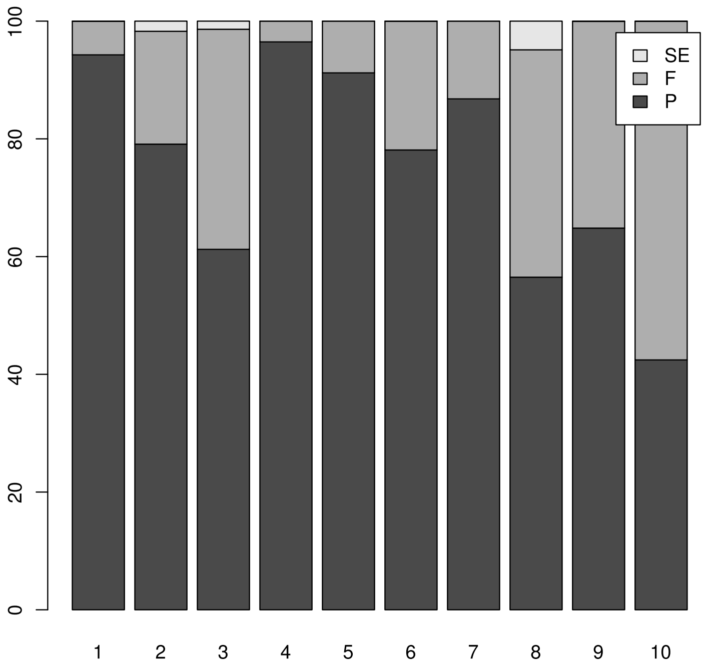


Figure 3.3: The outcome for the projects in terms of exceptions are low, which may rely on several conditions. Firstly, the test code coverage ratios are low, which may make the test cases inept of detecting failures. This is also suggested by this graph. Since fault are introduced by mutation operators, as with security exceptions, we can not expect the test classes to discover all potential security exceptions.

3.3.2 RQ2

What are the differences in proportions regarding P/F/SE for each mutant (class with mutant operator applied) among the different projects?

Mutation testing employs test cases for execution. A test case employs at least one class for each execution. For each execution of a test case, at most one mutation operator is applied to at most one class from the set of classes encompassed by the test case. The result from this test execution (P/F/SE) is attributed to **that** mutated class. To justify this, recall when the pre-mutated test case is executed, the test case yields P. When the outcome is different after a mutation operator is applied, the modification of the test case is represented solely by the mutant. The purpose of this is to see if a particular class in the test case, when mutated, is prone to produce an overweight of a specific outcome after a mutation operator is applied. This is accomplished by inspecting the *ratios* of *passed*, *failed* and *security exceptions* from the total outcome, w.r.t. the *mutants* grouped by project.

Ratios w.r.t. mutant:
Given a class $\mathbb{A}$ and a test case τ. Let Σ be the set of classes encompassed by τ, i.e. the classes that are executed during an execution of τ. Let $\mathbb{A} \in \Sigma$ and n, i, j, $k \in \mathbb{N}$.

Assume that in the case of $\mathbb{A}$, there are n variations, i.e. n mutants eligible to $\mathbb{A}$. This requires n executions of τ, i.e. one execution for each mutant of $\mathbb{A}$ to ensure that all mutants eligible to $\mathbb{A}$ are executed.

For each of the m executions of τ, we replace $\mathbb{A}$ with a different mutant and we record the outcome. Assume that τ yields i passed test executions (P), j failed test executions (F) and k executions where a security exception is thrown (SE).

For mutants, we then define the **p-ratio** as $\boldsymbol{(i \div n) \times 100}$, the **f-ratio** as $\boldsymbol{(j \div n) \times 100}$ and the **se-ratio** as $\boldsymbol{(k \div n) \times 100}$. Note that $i + j + k$ is always equal to n.

In this model, the outcome from τ is grouped by the mutants it encompasses (See Figure 3.4 for the relationship.)

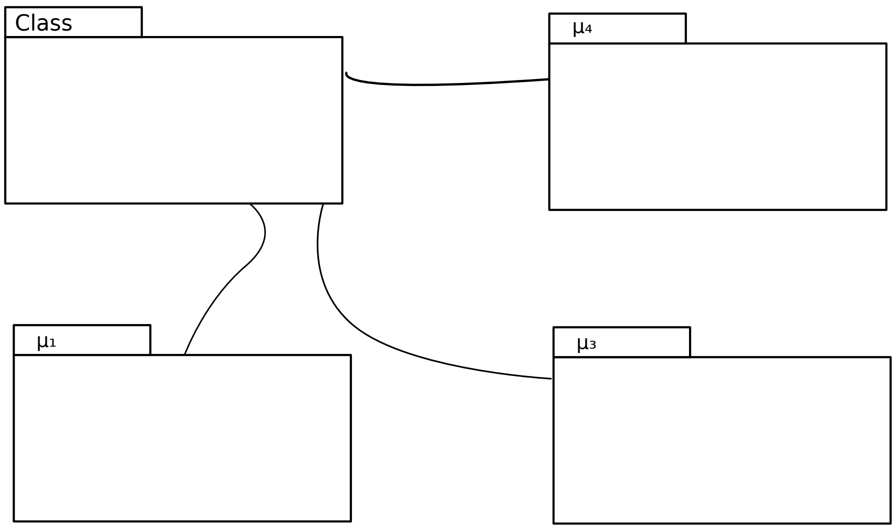


Figure 3.4: The class-mutant relationship. When calculating the ratios w.r.t mutant, we are interested in the yield (P/F/SE) of each mutant spawned from each class.

All ratios from all executions are collected and grouped by project. The type of ratio (p/f/se) is used when we divide the ratios per project into three groups. These groups will be visualized with box plots. Figure 3.5 visualizes the p-ratios, Figure, 3.6 visualizes the f-ratios and Figures 3.7 and 3.8 visualize the se-ratios for all projects with and without outliers. The solid line denotes the *mean* and the dotted line denotes the *median*.

Each point in the figures represents a ratio of either outcome for a mutant encompassed by the test case. This will enable us to see the proportions and other characteristics of the ratio distribution among the projects.

For instance, the most extreme outlier for project 6 in Figure 3.5 represents a class with a p-ratio of approx. 10. Likewise, the f-ratio is approx. 90 for the most extreme outlier in Figure 3.6. It is easy to see that the first Figure are almost the inverse of the second.

From Figure 3.5 we see that most of the projects contain classes with a high p-ratio. Not surprisingly, Figure 3.6 convey that the same projects contain classes with a low f-ratio. The outliers represents classes that are more prone to fail (or less prone to pass) after a mutation operator is applied.

Schuler et al. [23] partitions mutants into three categories: *Not covered*, *covered not killed* and *killed*. In Figures 3.5 and 3.6, *not covered* and *covered, not killed* applies to the first and *covered, not killed* (henceforth *live*) applies to the second.

The distribution from our executions is not completely unexpected. The same authors reported a distribution of 52 percent live mutants and 48 percent killed mutants. An analysis of our results shows that the overall ratio of passed mutants are as high as 95 , where the F-ratio is about 4. Taking account for the low coverage of the test cases for our test subjects, we justify our vast overall P-ratio with the quality of test cases which is suggested to be correlated with test code coverage [45].

Our data does not encompass information about mutants *not covered* by test code, hence we have no possibilities to say anything about the ratio of mutated classes yielding P due to inadequate test code coverage. The same authors report that 32 percent settled in the category *not covered*. We expect this ratio to be substantially higher for our projects. Our test code coverage is substantially lower than theirs, which had a total coverage ratio ranging from 77 to 99. This justifies our distribution further.

Furthermore we can see that the medians of some boxes in Figures 3.5 and 3.6 are departed far from each other. Just by looking at these figures, especially the median we can tell that the outcome from each project has a strong statistical difference. Also notice the overall median compared to the median for each project. This will lead to reliable results due to the difference among the projects.

To further justify this, we employ the non-parametric *Kruskal-Wallis* test. This is done with the statistics tool R.

We start with the vectors $\mathbb{P}_n$, $\mathbb{F}_n$, $\mathbb{E}_n$, $\{n \in \mathbb{N} \mid 1 \leq n \leq 10\}$, where n denotes the project. Each of these vectors contains the p-ratios, f-ratios and se-ratios respectively for project *n*.

The vectors $\mathbb{P}_n$, $\mathbb{F}_n$ and $\mathbb{E}_n$ are then passed to the R-method `kruskal.test()` *individually*. This means that we do a separate calculation w.r.t. each class of ratio, i.e. for each $\mathbb{P}_n$, $\mathbb{F}_n$ and $\mathbb{E}_n$ the calculation is performed.

`kruskal.test()` will yield the χ^2 and the *p-value* (Table 3.4). A low p-value indicates a high probability that a distribution contains statistically different distributions. This correlates with the observations made by looking at the figures and justifies our claim that the data has a strong statistical difference.

Table 3.4: Kruskal-Wallis test for outcome per class.

Per Class			
P	χ^2=275.44	df=9	p-value$<2.2 \times 10^{-16}$
F	χ^2=275.42	df=9	p-value$<2.2 \times 10^{-16}$
SE	χ^2=211.95	df=9	p-value$<2.2 \times 10^{-16}$

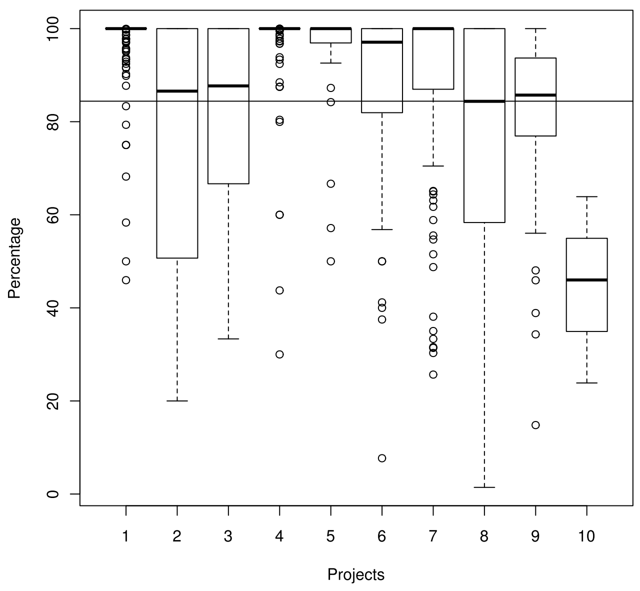


Figure 3.5: Test cases passed per class. The data for most test subjects does depart far from the true mean. Results for projects 1 and 3 are presumable due to badly written test cases.

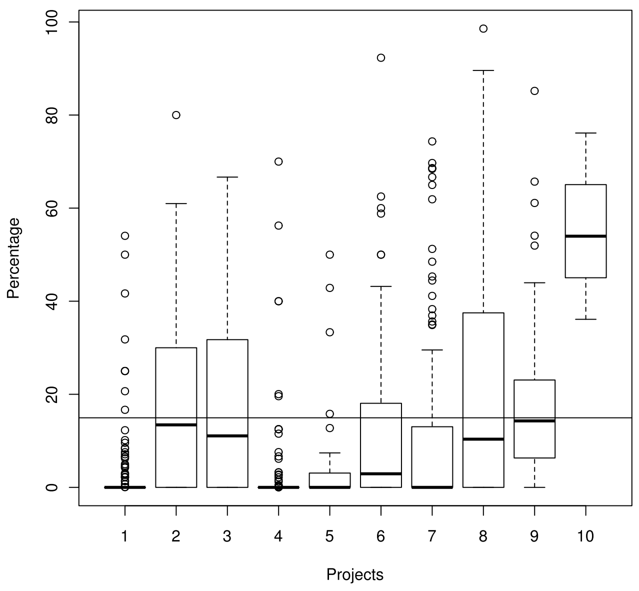


Figure 3.6: Test cases failed per class. This graph is approx. the inverse of Figure 3.5

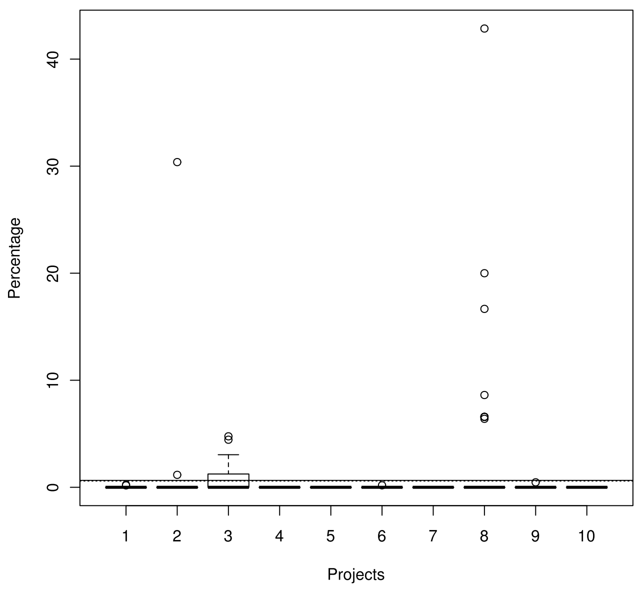


Figure 3.7: Test cases with security exception per class. The outliers are expected and are due to different responsibilities of the assessed class. These mutants are more prone to security exceptions because they contains business logic for file operations.

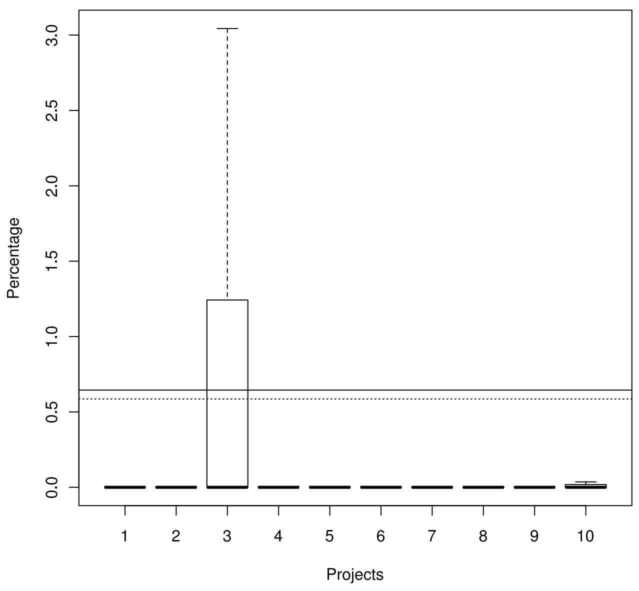


Figure 3.8: Test cases with security exceptions per class w/o outliers. Most of the se-ratio is contained in the outliers (Figure 3.7, hence the boxes are small for projects 2 and 8.

3.3.3 RQ3

What are the differences in proportions regarding P/F/SE for each mutation operator among the different projects?

To approach this, we start by using the same data foundation which was used to answer RQ2. The difference is the grouping of ratios. Instead of grouping the ratios by mutant, we group by the mutation operator and do a count of all classes which this mutation operator is applied to. The calculation of ratios is becoming a bit different:

Ratios w.r.t mutation operator:
Given a test suite Ω and a mutation operator μ and $\{m, n, i, j, k\} \in \mathbb{N}$.

Let Σ be the set of all classes encompassed by Ω eligible to μ, and Δ be the set of mutants created by applying μ to the elements in Σ.

Let σ_m be the elements of Σ and δ_n be the elements of Δ. Since μ can be applied to each σ more than one time, and each application produces a different δ, we have that $\|\Sigma\| \leq \|\Delta\|$.

Let $\|\Delta\|$ be m. After Ω is executed, we record the outcome for each of the m executions, each encompassing a different δ.

Assume that Ω yields i passed test executions (P), j failed test executions (F) and k executions where a security exception is thrown (SE).

For mutation operators, we then define the **p-ratio** as $\boldsymbol{(i \div m) \times 100}$, the **f-ratio** as $\boldsymbol{(j \div m) \times 100}$ and the **se-ratio** as $\boldsymbol{(k \div m) \times 100}$. Note that $i + j + k$ is always equal to m.

In this model, the outcome from the execution of the mutated classes are grouped per mutation operator (See Figure 3.9 for the relationship.)

All ratios from all executions are collected and grouped by project. The type of ratio (p/f/se) is used when we divide the ratios into three groups. These groups will be visualized with box plots. Figure 3.10 visualizes the p-ratios, Figure, 3.11 visualizes the f-ratios and Figures 3.12 and 3.13 visualize the se-ratios for all projects with and without outliers respectively. The solid line denotes the *mean* and the dotted line denotes the *median*.

Each point in the figures represents a ratio of either outcome for a mutation operator applied to a class encompassed by the *test suite*. Recall that we group by the mutation operator, which is common to the test suite. This will enable us to see the proportions and other characteristics of the distribution of ratios.

Again, by looking at the figures, we see variance in the data. This suggests a strong statistical variation among the projects. Also notice the overall median compared to the median for each project. We will also employ the Kruskal-Wallis test to support our claim about the variation.

We start with the vectors $\mathbb{P}_n$, $\mathbb{F}_n$, $\mathbb{E}_n$, $\{n \in \mathbb{N} \mid 1 \leq n \leq 10\}$, where n denotes the project. Each of these vectors contains the p-ratios, f-ratio and se-ratio respectively for project n.

The vectors $\mathbb{P}_n$, $\mathbb{F}_n$ and $\mathbb{E}_n$ are then passed to the R-method `kruskal.test()` *individually*. This means that we do a separate calculation w.r.t. each class of ratio, i.e. for each $\mathbb{P}_n$, $\mathbb{F}_n$ and $\mathbb{E}_n$ the calculation is performed.

`kruskal.test()` will yield the χ^2 and the *p-value* (Table 3.5). A low p-value indicates a high probability that a distribution contains statistically different distributions. This correlates with the observations made by looking at the figures and justifies our claim that the data has a strong statistical difference.

Table 3.5: Kruskal-Wallis test for mutants shows a strong statistical variance which suggests good reliability of the results.

Per Mutation			
P	χ^2=125.22	df=9	p-value<2.2 x 10^{-16}
F	χ^2=120.08	df=9	p-value<2.2 x 10^{-16}
SE	χ^2=138.88	df=9	p-value<2.2 x 10^{-16}

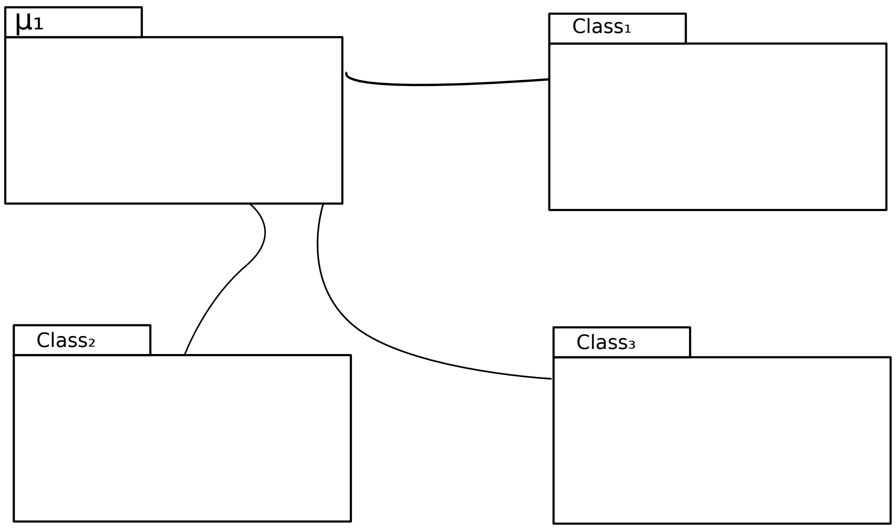


Figure 3.9: The relationship between the mutant operator and its eligible classes. When calculating the ratios w.r.t. the mutant operator, we are interested in the yield for each class in the test suite for which μ is applied.

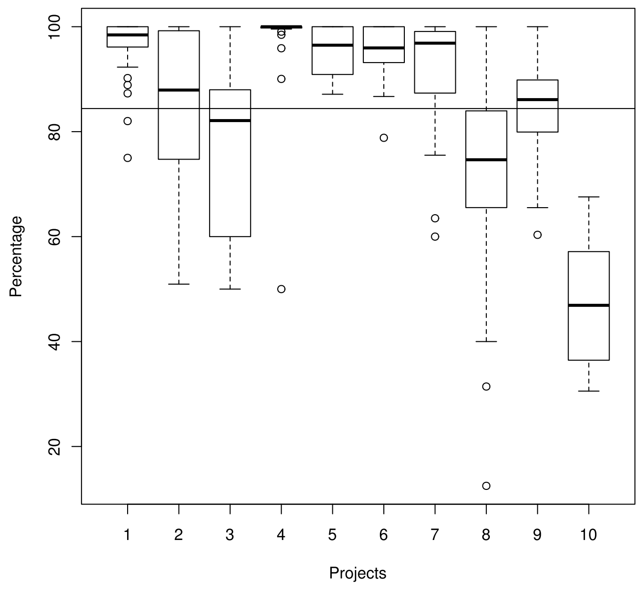


Figure 3.10: Test cases passed per mutation. Each point in each box represents a result from a test case executing a mutated class.

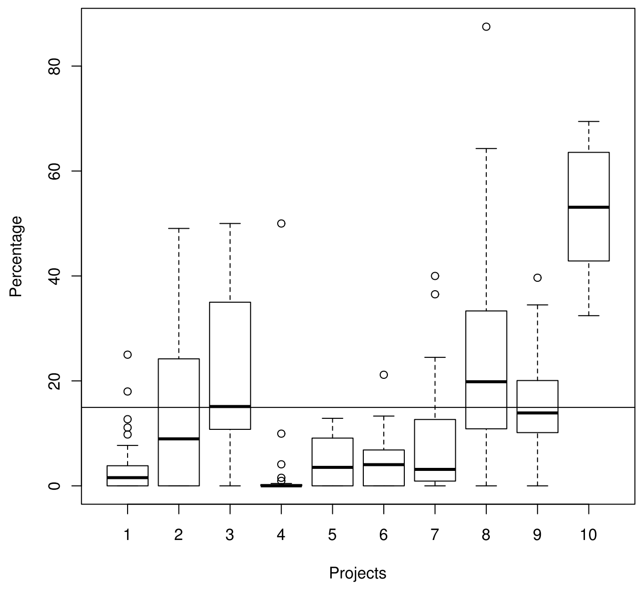


Figure 3.11: Same as for 3.10, but with failed test cases

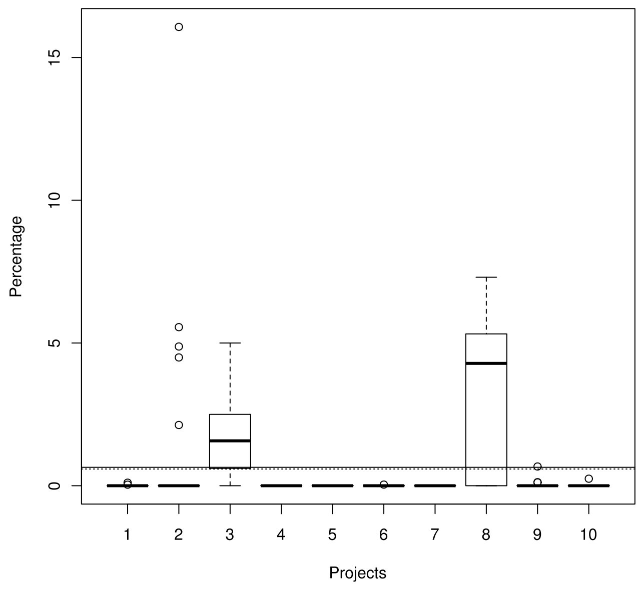


Figure 3.12: Test cases with security exception per mutation operator. Outliers are interesting, as they suggests single operators which causes many security exceptions. We expect to find important operators among these outliers.

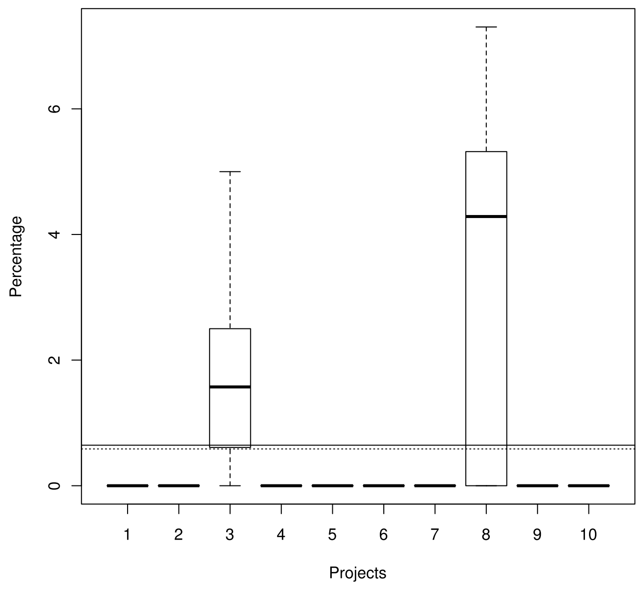


Figure 3.13: Test cases with security exception per mutation operator w/o outliers. Project 3 possess a good distribution and the entire se-ratio is related to file violations. We expect to find important mutation operators among this set.

3.4 Mutant Operators and Security Exception Distribution

To revisit the main objective of the study, we remind you that we initially wanted to assess whether or not mutation operators when applied to classes encompassed by a test case can be inclined to cause undesirable side-effects, such as malicious file deletions. These file deletions represents a potential risk, since deletion of files that are vital for the OS may cause the host to crash.

We have identified some of the mutation operators to be more crash prone than the other mutation operators. To produce Figure 3.14, we considered the mutation operators that when applied to classes made the class cause a file access violation. Other violations are omitted, since the potential risk is related to malicious file accesses. It is easy to discover a trend here, we can see that the frequency for *EAM* is twice as high as the second highest. EAM is an abbreviation of *Accessor Method Change* and makes modifications to object oriented code constructs, this is it an OO-mutation operator. A description of this is found in [39]:

The EAM operator changes an accessor method name for other compatible accessor method names, where compatible means that the signatures are the same. This type of mistake occurs because classes with multiple instance variables may wind up having many accessor methods with the same signature and very similar names. As a result, programmers easily get them confused …

An accessor method is a crucial part of the *object oriented encapsulation paradigm.* It provides a controlled access to a class or objects private fields, thus eliminating the need for a pathological coupling [40, ch. 4.4] between components.

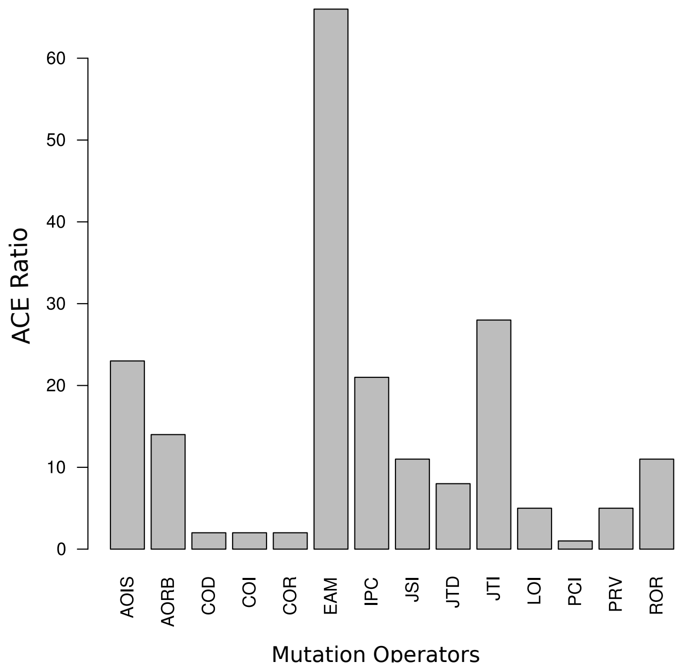


Figure 3.14: Distribution of the exceptions over all mutation operators that cause ACE related to file accesses. Other causes, like socket-exception are omitted. The *EAM*-operator is identified by the relative frequency of ACE. *AOIS* and *JTI* are also identified as operators with high probability of causing ACE.

An ACE from a modification by EAM sounds plausible because an accessor, as its name suggests, accesses a field and return its value to the caller. When wrong accessor is called, due to a programmatical error or mutant applied to code, an unexpected value will be returned. If this value is to be used for file access, the incorrect file will be targeted. It is difficult to say anything general about the possibility that a file vital to the OS will be affected as a consequence of this mutation operator, however the possibility may exist (or it may not exist.) As an example, we provide a constructed code segment which is prone to cause an ACE when EAM is applied.

Consider listing 3.3, a fictive class that stores information used for the persistence sub system for a (also fictive) program. When an object is constructed, it is passed two parameters; the name of the main database file[2] and a temporary file. When a component which requires this information is executed, it will query an object of this class for file names to use.

Imagine that the string *database* exists from a recent session and data processed by different sessions are accumulated in the file that it represents after being processed. When a component passed this object is requesting the *intermediate* (temporary) file name, it will typically call *getIntermediate()* and continue processing, and maybe delete the temporary file after the process is complete. Now, if it instead called *getDatabase()*, it will overwrite the main database and delete.

Listing 3.1: FileLibrary (constructed)

```java
public class FileLibrary {

    public FileLibrary( String database, String intermediate ){
        this.database = database;
        this.intermediate = intermediate;
    }

    public String getDatabase() {
        return this.database;
    }

    public String getIntermediate() {
        return this.intermediate;
    }

    /*
    * Fields for storing names of
    *  - Database-file
    *  - Temporary work file
    */
    private String database;
    private String intermediate;
}
```

The EAM-mutant mimics this error pattern by altering the code in this fashion: (Notice that, for illustrating purposes **two** mutants are added to **one** class file since two methods with equal signatures cannot co-exist.)

[2]Flat files are rarely used for databases nowadays, just for illustration purposes.

Listing 3.2: FileLibrary (constructed)

```java
    public String getDatabase() {
        return this.database;
    }

    public String getIntermediate() {
        return this.intermediate;
    }
```

$$(line\ 9)\ getDatabase() => getIntermediate() \tag{3.1}$$

$$(line\ 12)\ getIntermediate() => getDatabase() \tag{3.2}$$

Listing 3.3: FileLibrary with EAM (constructed)

```java
    public String getIntermediate() {
        return this.database;
    }

    public String getDatabase() {
        return this.intermediate;
    }
```

Indirectly, an application of 3.1 and 3.2 will mimic an error done by the programmer as described above. The semantics however is equivalent. Deleting a main database file is not necessarily disastrous to the OS, but will definitely cause problems.

Regarding the histogram 3.14, the mechanisms that induces the ACEs are not known, let alone the point here. The significant issue is that they were provoked by EAM, which is an important discovery in our study. For examples of *JTI* and *AOIS* refer to Section 2.6.1 and 2.7.2 respectively.

3.4.1 Mutation Operators with a Higher Probability of causing Side Effects

We have identified the mutation operators *EAM*, *JTI* and *AOIS* to be the three most important mutation operators discovered from our study because of their frequency of provoked ACEs. Whether these in general, when applied to a class, makes the class prone to perform undesired side effects as unintended file deletions is difficult to say. To make such a suggestion, we need to consider the characteristics of the test subjects as well, e.g. frequency of file accesses, type of accesses *(read, write, execute and delete)*.

Chapter 4

Discussion

4.1 Mutants and Evidence of Hazards

In the process of mutation testing, we have discovered that some mutation operators can cause undesirable side effects to Java programs. These are file accesses that were not performed by the program originally. Three mutation operators are identified by the characteristic that they are more prone to make a program cause these undesirable side effects when applied. These three are the *EAM*, *AOIS* and *JTI* operators and are selected on the basis of the frequency of file access violations caused by the class for which they are applied.

While there is no hard evidence that any mutation operator causes a class to delete vital files, one should be careful to omit this possibility. We have already seen several live examples on file accesses that were different from a pre-mutated program compared to a mutated program.

In general, for a class to be prone to perform these side effects when a mutation operator is applied, it needs to contain business logic for file operations. Projects that utilizes files for e.g. data persistence are more prone to experience these side effects. From this, we suggest that when performing mutation testing on projects involving quite a few file accesses, one needs to take account for the side effects of applying the mutant operators we identified.

The root causes of these side effects are quite a few times due to modifications of variables and fields in the Java classes, where these fields are used for storing file names that are intended for later file operations directly to the disk.

When this experiment was just an idea, we had no knowledge of unwanted side effects in the fashion as described here. As more of the test executions of the projects were accomplished, the idea became evident as we observed file access violations.

The potential hazard of applying these operators depends on quite a few factors, e.g. the nature of the program.

All non-trivial program employs some sort of data persistence, be it on database level or file level. Roselli et al. suggests a high file system workload in production environments [56]. Since every client in such an environment is using software, it is evident that these accesses is commissioned by some software. This suggests that almost any system which is subject to mutation testing may suffer from side effects.

It is suggested that it is as much variation in source code as there are programmers [41, p. 872], each programmer possessing different imagination and habits [67, p. 119]. Oman and Cook are even suggesting a taxonomy for programming style [53]. Programmers are inclined to implement "interesting" solutions to different challenges, which may deviate from standard *design patterns.* A design pattern is a general reusable solution to a commonly occurring problem in software design. Deviating from these design patterns may introduce errors which in turn will make the code less robust and prone to errors. This justifies the arcane code excerpt in

Listing 3.3. Since we may encounter similar constructs when performing mutation testing, we suggest that one may also encounter hazardous side effects for that cause.

4.2 Custom Security Manager and Application Domains

Several approaches has been proposed to meet the challenge of confining software executions in a sandbox environment. Sidiroglou et al. proposes a system called ASSURE [62] which *introduces rescue points that recover software from* **unknown** *faults while maintaining both system integrity and availability.* Unknown faults are caused by errors not revealed by a system test, which may be abundant even after a thorough testing process. Their aim is to trap these faults and continue running by restoring the systems internal state to a recent working state.

Liang et al. proposes a *safe execution environment* called *Alcatraz* [36], *An Isolated Environment for Experimenting with Untrusted Software.* This enables developers, according to the papers to *try out new software or configurations changes to existing software without the fear of damaging the system in any manner.*

Yet another approach is proposed and implemented by Kiriansky et al. [34], a framework employing *program shepherding*, which is capable of *preventing executions of malicious code ... monitor all control transfers to ensure that each satisfies a given security policy.*

A summary of these three approaches is that the first is allowing any operations and retreats to a earlier restoration point when a malicious operation occurs. The second provides a sand-box that isolates an executing program and confines resources that are utilized. The third provides high granularity of restrictions by only allowing the program under execution to access a subset of the instruction set architecture and interface provided by the OS.

A mode for these three is low overhead while executed in conjunction with the subject program, according to the papers. Note that the three aforementioned framework are intended for a more general usage, e.g. *experimenting with untrusted software*, *prevent security attacks* and *protect the host against unforeseen failures caused by software failures.*

When a Java program is executed and the CSM is deployed with a security policy adapted to the execution cycle, it reflects one property of *Alcatraz* and the *program shepherding* paradigm. All transfers of control are intercepted before it reaches its destination and checked to see if it satisfies the given security policy.

Just as for *ASSURE*, all these operations are logged which enables the software tester to do an error trace such that malicious operations can be weeded out. CSM is an experimental component initially used for the purpose of trapping violations by adhering to a security policy, thus we do not expect it to compete with any of the three frameworks. Nevertheless, we believe from what we have seen that it may serve the purpose of an extra layer of security when running automated mutation tests.

We do not expect issues of this idea to be absent. One important issue is that the CSM will interfere negatively with the performance of the test subject. Papers report an CPU execution time penalty from 5 percent to a whopping 100 percent per resource access statement [25]. The same report discourages the use of any security manager for applications that (e.g.) utilizes frequent SocketPermission requests for many different hosts or if the policy files are huge and the application is restarted too frequently (This is the scenario for our experiment.) See Section A.1 for more detailed information on the CSM and performance. These characteristics will increase total execution time by adding a start up penalty for each execution.

The developers of *Jumble* [29] states that their rationale is to perform mutation testing in conjunction with the rapid build cycles carried out by their development environment. The code is checked out every fifteen minutes and built, tests are run until a new checkout occurs, which will cause the testing process to restart. Tests that are accumulated are put in a queue and will be processed during idle time, typically during the night (where no modifications in code occurs.) The critical issue is to get sufficient speed of the mutation testing, hence mutation is performed at byte code level.

There is no doubt that a deployment of the CSM will decrease the performance for a mutation testing process. When performing unit testing, every test case requires a start-up for each execution. We know that in the context of mutation testing, a start-up is required for every mutant encompassed by the test case. For this reason, we will not recommend deployment of the CSM when execution speed is crucial.

When source code is modified, the policy files also needs an update. Recall that the policy files are created for each test case by executing and recording the output from the CSM once. Since we have developed a script that automates this process, the recommendation in [25], which discourages the use of any SM if *...maintenance of policy files are cumbersome (due to e.g. code changing often, many different jar-files,etc)* is considered invalid.

When a unit test which violates any permission is encountered, it will terminate and the outcome will be logged, then the next unit test will commence. Assuming that code in the unit test will affect the system availability, the software testers must decide whether they should be content with a slower execution or a potential system hazard.

For these reasons we will not recommend the CSM for mutation testing processes where performance is crucial, but will suggest that it may be useful in mutation testing scenarios where side effects have a high probability to occur.

Chapter 5

Threats to Validity

Since there is no such thing as perfect data and perfect analysis that would yield 100 percent reliable results, there is a need to identify the threats to validity.

In this chapter we will discuss the *realism* in the means of *realistic tasks*, *realistic subjects* and *realistic environment* [63].

The first criterion is concerned with *the size, complexity and duration of the involved tasks.* Basically, this means that an observable effect must be revealed from our test subject when execution is performed in the same fashion as it is in a real-life environment, e.g. the industry.

The second is concerned with the selection of subjects to perform the experimental tasks, i.e. in what extent does the subjects represent the population that we wish to make claims about?

The last is concerned with the subjects and its tasks and if they are carried out in a realistic manner. Is the experimental environment configured with a supporting technology that resembles and industrial development environment?

Furthermore, we discuss validity of the results of our experiment in the context of three different types of threats: *internal*, *external* and *construct validity* [11, p. 13] and [1].

The term *realism* is related to the concept of *external validity.* We choose to include an evaluation of the realism together with the three means of validity, we believe that realism will provide the reader with a more detailed perspective.

5.1 Realism

Sjøberg et al. [63] discusses *mundane realism* which *refers to the resemblance of an experiment with read world situations and, therefore, with our ability to generalize the result of the experiment to industrial practice.*

5.1.1 Realistic Tasks

We have already seen that mutation testing is becoming more popular as a test technique [30] and that the extent of use is rapidly increasing. Mutation testing is also widely used in conjunction with unit testing, JUnit in particular, which is a framework adopted by the industry. The implementation of the test framework that utilizes mutation testing varies, but the mode is the same disregarding if mutation testing is executed manually by human software testers or automated by e.g. a build-server.

By virtue of this, we conclude that our tasks are realistic. We are jointly utilizing well-proven paradigms to produce our results (unit testing in conjunction with mutation testing), which is a good premise for achieving valid results.

5.1.2 Realistic Subjects

Our experiments consists of software projects downloaded from SourceForge and are widely used either as stand-alone programs or intricately coupled to other frameworks (e.g. BCEL, which is the core of Jumble [29].)

Despite that all these programs possesses different qualities regarding file accesses, we consider these as realistic subjects due to their life cycle span, maturity and utilization.

5.1.3 Realistic Environment

In a production environment, many processes are interconnected and there is a lot of synergy. A test environment should resemble a production environment as much as possible. The scenario of the production environment makes it difficult to predict how a programming systems product is behaving when it is released. The ideal scenario is to have a testing environment resembling the production environment as close as possible.

When performing unit testing, the scene is a bit different, each test case is a stand-alone entity with methods for setting up pre-conditions and proper scaffolding for proper execution.

In this study, we aim to contribute to make the testing process safer by eliminating potential hazardous mutation operators from MT processes. We do that by running JUnit test cases with pre-mutated projects and mutation operators as input. We see no obstacles for an extrapolation of this technique into real-world scenarios, hence we believe that the rules which are valid for our study can be adopted for general usage.

5.2 Validity

5.2.1 Internal Validity

Internal validity concerns the cause and effect of variables. We wanted to investigate if such a path existed between the mutant *(independent variable)* and malicious file operations, the *(dependent variable)*. Our test cases are deterministic, as we have observed that the same results are produced for consecutive executions of test cases. Likewise, the environment is unchanged for each and every execution.

We encountered problems which caused spurious effects (see Section 6.4). They were eliminated and the test subjects were reassessed. This is supporting the integrity of the internal validity.

A topic for discussion is the *quality* of the test cases (written by the software developers.) From table 3.2, we see that the *mutant kill rate*, which is known to measure the effectiveness of a unit test, varies. Likewise, the coverage also possess a strong fluctuation. As this is correlated to test case effectiveness, this also gives a good indication of the test suite quality.

Nonetheless, papers [13, 70, 71] suggests that a test suite should cover approx. 80 percent of the code to be considered a test suite with *good reliability.* With an average coverage of 28 percent, this does not speak in favor for internal validity. It is reasonable to assume that more mutants could have been killed with a higher coverage ratio. From a higher kill ratio, a higher ratio of discovered hazardous operators follows.

Another consideration are the flaws in MuJava i.e. it cannot compile classes with generic code constructs. Since instrumentation is performed on source code level when object oriented mutants are applied, the source code needs to be compiled in order to be tested. The error messages issued by MuJava addressed generics and compilation errors quite a few times, as there are a lot of classes with generic code constructs. This is expected, as many of the latest version of the projects were utilized in our study. For this reason, we take into consideration that quite a few classes were omitted for this reason.

It is also expected that the amount of file operations will fluctuate when different programs are executed, i.e. some programs employs file system operation more than other does. This will of course affect the results, as for some programs no file access violations were detected. A more appropriate set of test subjects, selected with bias to frequent file accesses might have improved our foundation.

How does this affect the total outcome from the assessments? This is difficult to say, we do not possess adequate information about the omitted classes to answer that. But we can be certain that the internal validity is threatened by this flaw in unknown extent.

5.2.2 External Validity

The main result from this study are a set of mutants considered hazardous. Despite threats to internal validity, we suggested that three operators can impose a potential hazard to the testing process; *EAM*, *JTI* and *AOIS*. Of course, more studies resembling this should be conducted and evaluated to support our suggestion before any results are deployed into existing mutation testing systems.

Nonetheless, if some testers are inclined to solve problems before they arise and apply these results before further research is available, they should consider some issues regarding the importance of operators to a test cycle: Studies are conducted on optimizing the efficacy for mutants by optimizing the subset of the most important operators [47,61,64]. The latter paper considers the MuJava mutation operators and they report that *EAM* and *AOIS* are among the top three regarding spawning of mutants. They suggest that omitting these comes with a cost of less mutants killed, which implies a deteriorated set of mutants which are prone to cause the unit test to trigger.

For these reasons considered alongside with the only a theoretical possibility that these mutants can cause malicious file deletions, we are certain that the external validity has been compromised.

5.2.3 Construct Validity

The basis for this experiment is to measure the cause-effect relationship for the outcome of a test case when a mutation operator is applied. This is exactly what we have done. The only variable during the execution of the study is the class which a mutation operators is applied to. Several executions with the same parameters did not reveal any spurious relationships regarding the cause-effect relationship between mutants and its outcome. From these observations we justify the construct validity.

Chapter 6

Implementation: Challenges and Obstacles

To harness the foundations for this study, which basically are log files from executions of JUnit test cases from different software projects, a methodical approach is required. All projects should be executed under equal conditions, hence there was a need for a framework that would assure that each and every project is initialized and executed on equal premises. To meet this challenge, several strategical choices were taken.

6.1 Execution Platform and Storage Capacity

As we know, during mutation testing, an enormous amount of mutants can arise from just a few lines of code. When mutating code with megabytes of source code, we expected almost an innumerable[1] amount of mutants. Take BCEL (1) for instance; the pre-mutated byte code tree comprising 411 classes with a total size of 2.3 megabyte. After all feasible mutants were applied, the mutant tree (byte code tree with mutants) contained 43957 class files requiring 1.2 gigabyte of storage, overhead of the file system not included. Another example is Apache Commons Math (7). Pre-mutated, 628 classes with a size of only 3.5 megabytes exists. After applying mutants, over 420 000 mutants were generated with a size of 3.5 gigabytes. In neither of these space requirements calculations, the *space occupied* is calculated. This number is presumably larger due to file system overhead. With a total of ten projects subject to the same procedure, several gigabytes of storage is required.

6.2 Execution Platform and Processing Capacity

We also know that mutation testing is an expensive process, alluding to Section 1.4.2. To get an idea of the total running time for all ten projects, an analysis of Squirrel SQL (1) is performed w.r.t. find the *execution time for a single mutant* (table 6.1). When we mention *speed* in the context of execution, we are referring to $mutants \times s^{-1}$.

Data from Squirrel SQL (1) shows that the execution speed is $\mu^* = 2.54$ mutants $\times\ s^{-1}$. The σ is higher than expected due to the three outliers with execution time from {180 - 1541} seconds. When assessing 420 000 mutants, we expect a total execution time of approx. 420 000 $\times\ s^{-1} \times 2.54$ s ≈ 296 hours = 12.3 days. Alas, this is only applicable if the process is allowed to run continuously without any interruption. As we soon will elaborate, the process was not straight-forward, let alone unperturbed.

[1]Not in the context of theoretical mathematics.

To accelerate the speed of the testing process, sessions were run in parallel. The execution platform consists of high-end servers having 8 cores each, thus assumed capable of running several processes simultaneously without affecting the throughput of the other processes. Given two mutant execution processes $\mathbb{A}$ and $\mathbb{B}$, no quantifiable drop in speed were detected when either were executed in parallel vs executed in solitary. With this considered, we ran two processes in parallel on each of the four equal equipped servers available at that time.

Table 6.1: Mutants and their execution time, gathered from the execution of 47730 mutants from Squirrel SQL, $\mu = 2.54$ and $\sigma = 7.37$. The high σ is caused by the outliers in the last tree rows.

Time consumption (s)	#mutants
1	3275
2	23656
3	17606
4	2108
5	171
6	24
7	527
8	317
9	21
10	8
11	5
12	3
13	1
16	1
180	5
181	1
1541	1

From this table, we can expect that the average execution time for *one* test case to be less than 3 seconds, which will aid us when total execution time is requested.

6.3 Choice of execution platform

From the inducements above, we chose to arrange the testing environment on the hardware at USIT[2]. Since the author is familiar with the technology they possess and the organization that administer them, we believe that it is possible to request modifications or more resources if required. In addition, they provide a superior help-desk for their users and the latency time for processing requests is fast. All these claims are experienced by the author.

6.4 Impediments related to the Execution Process

As stated earlier, the execution process did not carry through unperturbed. In the middle of an assessment of BCEL (1), the JUnit tests reported *java.io.IOExceptions* which *Signals that an I/O exception* [related to the test case] *of some sort has occurred.* The execution framework does not implement an early warning system which intercepts and reports failures during execution, thus the process was allowed to elapse.

[2]The University [of Oslo] Centre for Information Technology

This exception is processed with the modus operandi of 3.2, i.e. an unknown result is issued to the code responsible to the evaluation process for the results. Recall that this code accepts only *P*, *F* or *SE*. The default evaluation when none of these are entered is *F*. This incident was not discovered until a manual assessment of the log file was performed. The symptoms was an incomplete log file with IOException reported as the latest results. Further investigations were made by executing the last test case and the result was still en IOException.

Unfortunately (or fortuitously), this concerned other test subjects also. We then assumed that when two or more disjoint test subjects running on the same platform yielded the same exception, the error is compelled to be caused by factors not encompassed by the execution framework. This assumption proved to be correct; USIT provides a default disk-quota of 2 gigabyte per user account. The framework directs all logging to disk storage. IN addition, quite a few test cases persists data to hard drive storage as well. Then the quota was exceeded, the operations were rendered impossible by the quota-management mechanism of the platform. Java responded to this by throwing an IOException.

This problem was elucidated by requesting more disk space, which was obliged by USIT. Nevertheless, valuable time was dissipated and all processes needed to be restarted. However, no modifications were made to the framework, since the extra resources of 20 gigabytes of disk space was considered to be more than adequate for the storage requirement for proper execution.

Unfortunately, the problems did not cease to exist at that point. During execution, another exception were raised. Log files from several executions reported *java.lang.OutOfMemoryError*, which is *Thrown when the Java Virtual Machine cannot allocate an object because it is out of* [available] *memory, and no more memory could be made available by the garbage collector.*

The only usefulness of this scenario is that it concerned more than one of the test subjects. When investigating the cause of the exception, we ended up by monitoring the *process list* (the UNIX-command *top*) during execution.

During normal operation, at most two processes commissioned by the framework should co-exist per session. We discovered that almost 30 (sometimes more) processes were spawned from one single session, each utilizing valuable memory from the host. Each of these processes were stalled JUnit test cases, which execution time was prolonged considerably. This may be caused by the mutant specifically or the class that was prone to suspend execution when only small modification were made.

An investigation of the cause is beyond the scope of the experiment. In either case, the process is suspended, but live, thus consuming resources. Unfortunately, JUnit was not configured with a maximum running time (which when exceeded will cause a *TimeoutException*). By our modest knowledge of JUnit, the code needs to be instrumented if timeout should be configured. This instrumentation is only applicable for JUnit versions > 4. Some the our test subjects utilizes JUnit < 4, hence this problem requires a different solution.

To surpass this obstacle, a modification to the framework were presented. Still written in PERL and providing multiple possibilities, we chose a model that confined the sub process (spawned by the initial process) within a configured time frame. The super process (sometimes designated *mother-process*) accomplishes this by employing *fork*, which is a mechanism of the UNIX and Linux sub system. The new process is set up with an *alarm* (POSIX) which will send a *signal* which again will terminate the application after a given amount of seconds. This ensures a maximum execution time for every sub process, which again will prevent OutOfMemoryErrors caused by non-terminating sub processes. After modifications were deployed, the framework is considered modified. To ensure identical environment for all test subjects a restart of the entire assessment was required. This incident of course caused a significant increase in total execution time.

6.4.1 Sub Processes and Separate Memory Spaces

One disadvantage of this model is that data from the sub process, i.e. the result from test case executions cannot be shifted to the mother process without further ado. The sub process is an exact copy of the super process, but resides in a different location of the memory. Variable updates in either processes will not affect the other as these are completely disjoint. Trivial operations, such as populating a variable with data becomes tedious when separate memory spaces are utilized.

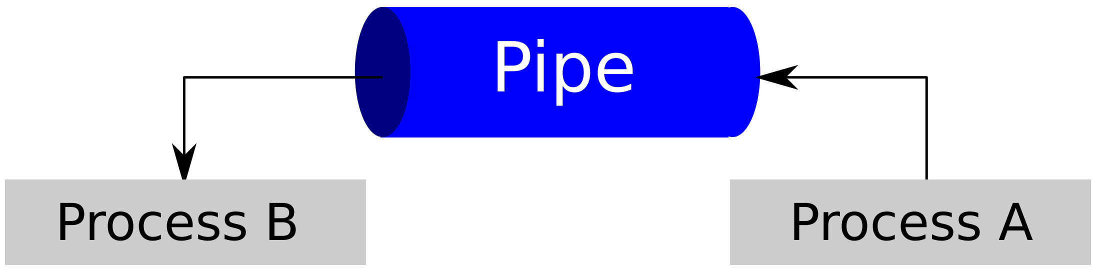


Figure 6.1: A UNIX-pipe makes it possible for two disjoint processes to share information. Data from process A is piped to process B.

One approach is to establish a pipe that is capable of reading and writing both memory spaces. The "openings" symbolizes entrance and exit which are visible to both processes, allowing them to pass streams of information between them. This implementation makes the solution concerning the separate memory space transparent and normal operation can continue. Most important, it alleviates the memory consumption of the framework which introduced a great antagonist.

6.5 Impediments related to Omissions of the Data Foundation Specification

Initially, we were logging just a subset of the information currently logged. The idea was to establish a book-keeping of all security exceptions and which mutation that was involved, which were considered the most important details. After a discussion with the supervisor for this thesis, a modification was proposed, This proposition included logging of the *exception message*, a description of the incident that caused a security exception. This enables us to dichotomize the security exceptions into security exceptions *related to file accesses* and *not related to file accesses*. Recall that the first group of security exceptions is believed to cause the test environment to become unstable by malicious file deletes.

The implementation of this facility was not considerably hard, but as always, a re-run of the test subject altogether is indeed required, due to modification of the framework. It is easy to understand that this incident rendered many hours misspent, thus more valuable time was wasted.

Chapter 7

Conclusion

This thesis reports on an empirical study performed on ten Java projects, which are available from Sourceforge. In these projects, faults were seeded by a widely used and mature Java mutation system, MuJava. We investigated the potential hazards that may be caused by Java programs that are subjected to mutation testing. This was accomplished by deploying a custom security manager in conjunction with standard mutation testing. The custom security manager will report every security exception from the test case and enable us to perform error-traces.

When evaluating the distribution of security exceptions among the different mutation operators, we only considered the cases where a file access violation was performed. Our results suggests that three mutants are more prone to make classes yield security exceptions than other mutation operators, hence we identified them as operators with high probability of causing hazards. These operators are *EAM*, *JTI* and *AOIS.* Because of these characteristics, we suggest that eliminating these from future testing sequences may improve the stability of the mutation testing process.

We also discussed the confounding effects to the mutation testing process which becomes evident when we omit the aforementioned operators from the set of mutation operators available to mutation testing. Two of the operators identified in this study are reported to be of the top 3 mutation operators based on effectiveness measured by the amount of mutants spawned when they are applied to a class. They are also reported to trigger test cases more than other mutation operators.

By removing these, we settle with a deteriorated set of mutation operators available, which will affect the mutation testing process negatively. Effectively, this means that a compromise must be made between a deteriorated set of operators versus improved stability of the testing process.

We also discussed other uses for the custom security manager and in which extent this could replace other suggested software security agents. This should initially be accomplished by intercepting the current operation performed and terminate the process of a violation occurs. The answer to this is that the overhead of executing this mechanism is immense, hence we cannot suggest other practical usages at the time of writing.

We also elaborated challenges met during development and execution of the framework utilized in this experiment, which caused the total time consumption of the experiment to elongate.

Appendices

Appendix A

Appendices

A.1 Why not Byte Code Mutation?

Mutation of code may be performed on *source code level* (source code mutation) or *byte code level* [29] (byte code mutation). The latter omits the need to compile every mutant that is generated, since the modification done *after* the source code is compiled. Mutation on byte code level is less time consuming than source code mutation since the overhead of compiling is reduced in the amount of n, where n is the amount of eligible mutations of a test subject.

When applying *method-level operators*, MuJava performs these on byte code level. This is assumed to be unproblematic since these operators do not alter code structures related to the object oriented paradigm. They mostly operate on arithmetic, logical and conditional operators. Remember that modifying the *access modifier* (private, default, protected or public) for a method adding the *static*-keyword are all object oriented features, hence left untouched by method-level operators. On the other hand, object-oriented mutation operators are applied on source code level. While the rationale for this is not mentioned in the sparse documentation for MuJava, we can contemplate several occasions for this. To illustrate this example, we utilize an imagined mutation system called *Byte Code Mutator* (BCM).

Firstly, when the test subject is compiled and resides in the byte code tree, BCM will load each and every class into memory. Then, the byte code is analyzed and mutants are applied by modifying the byte code. Consider Listing A.1 and the resulting byte code for OoTest (Listing A.2) which is where the mutant is applied:

Listing A.1: BCMTest (constructed)

```java
public class BCMTest
{

  public static void main( String[] args )
  {
    OoTest ot = new OoTest( "We␣will␣not␣use␣Hello␣World!" );
    ot.getOut();
  }
}


class OoTest
{
  public OoTest( String t )
  {
    out = t;
  }

  public void getOut()
  {
    System.out.println( out );
  }

  private String out;
}
```

Listing A.2: OoTest bytecode (constructed)

```
class OoTest extends java.lang.Object{
private java.lang.String out;

public OoTest(java.lang.String);
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: aload_0
   5: aload_1
   6: putfield #2; //Field out:Ljava/lang/String;
   9: return

public void getOut();
  Code:
   0: getstatic #3; //Field java/lang/System.out:Ljava/io/PrintStream;
   3: aload_0
   4: getfield #2; //Field out:Ljava/lang/String;
   7: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String
      ;)V
   10:  return

}
```

When applying the *JSI*-operator *(static modifier insertion)* to the class OOTest, the source code and the byte code is modified (Listings A.3 and A.4.)

Listing A.3: BCMTest JSI (constructed)

```
public class BCMTest
{

  public static void main( String[] args )
  {
    OoTest ot = new OoTest( "We␣will␣not␣use␣Hello␣World!" );
    ot.getOut();
  }
}
```

```
10
11
12 class OoTest
13 {
14   public OoTest( String t )
15   {
16     out = t;
17   }
18
19   public void getOut()
20   {
21     System.out.println( out );
22   }
23
24   private static String out;
25 }
```

Listing A.4: OoTest bytecode JSI (constructed)

```
1  Compiled from "BCMTest.java"
2  class OoTest extends java.lang.Object{
3  private static java.lang.String out;
4
5  public OoTest(java.lang.String);
6    Code:
7     0: aload_0
8     1: invokespecial #1; //Method java/lang/Object."<init>":()V
9     4: aload_1
10    5: putstatic #2; //Field out:Ljava/lang/String;
11    8: return
12
13 public void getOut();
14   Code:
15    0: getstatic #3; //Field java/lang/System.out:Ljava/io/PrintStream;
16    3: getstatic #2; //Field out:Ljava/lang/String;
17    6: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String
         ;)V
18    9: return
19
20 }
```

Now, go to line 17 and 16 in the byte code listings (Listings A.2 and A.4 respectively). Notice that `3:  getfield #2` is replaced by the command `3:  getstatic #2`. Likewise, line 10: `5:  getfield #2` is replaced by `5:  putstatic #2`.

Explanation follows:

- `getfield` *...gets a field value of an object objectref, where the field is identified by field reference in the constant pool index (index1 << 8 + index2)*
- `getstatic` *...gets a static field value of a class, where the field is identified by field reference*

in the constant pool index (index1 << 8 + index2)

This means that one mutant made several modifications to the byte code, and this is expected. The results are equivalent when running both code examples.

Now, contemplate a change in byte code which is only considering one line per mutation. This would change either line 3, 10 or 16/17, causing the class not to be verified by the class loader, thus it is not executed and the test case would fail. To circumvent this early in the process, we choose source code level mutation. Considering the overhead by compiling all source code files for each mutation vs the time available for this process, this is defensible. Especially since we do not know exactly the outcome of byte code mutation.

Another consideration is that we do not know if all test subjects utilized proper interfacing between objects, i.e. *object coupling.* If worst comes to worst regarding object coupling, the programmers may have objects accessing public scoped (or default scoped) fields in other objects, also known as *pathological coupling.*

While this is bad practice (site code complete), there is no guarantee that this is not present. When this coupling is present, a mutation would, if not caught by the class loader, render the result failed.

To ensure class coupling compatibility, we choose to make the compiler generate byte code from mutated source code.

A.2 CSM Performance

To support the overhead for the CSM, we measured 747 test case executions for Squirrel SQL.

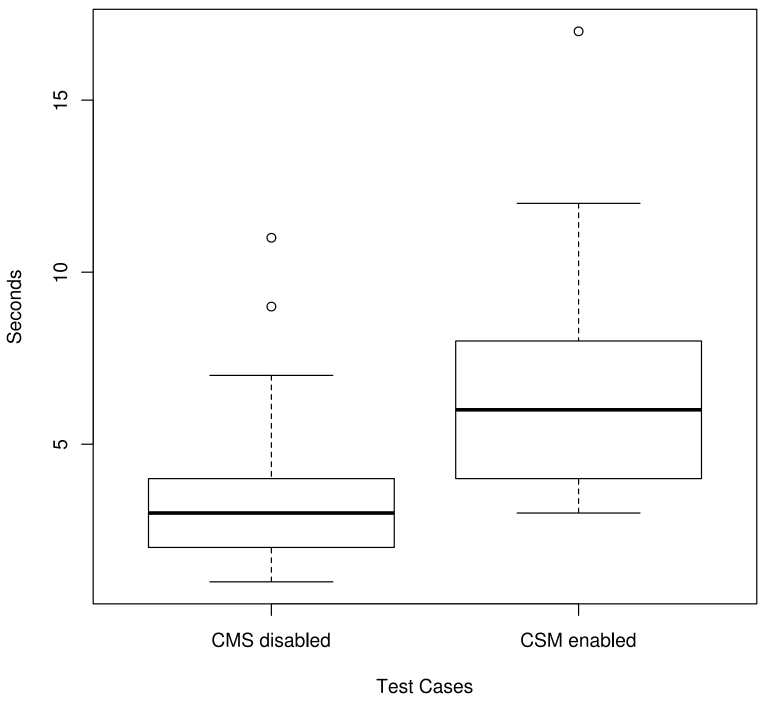


Figure A.1: It is easy to see the variations for the execution time when the CSM is activated.

From this data, we expect an average execution time of 3.34 seconds for each test case when the CSM is disabled and 6.02 when CSM is enabled. The standard deviation is low for both configurations, 1.99 and 1.32 seconds respectively. This supports Herzog et al. [25] and their reported performance of the Java Security Manager.

A.3 Critical Values of Correlation Coefficient (R)

XY-pairs	df	.01	.05	.1
3	1	0.988	0.997	1.000
4	2	0.900	0.950	0.990
5	3	0.805	0.878	0.959
6	4	0.729	0.811	0.917
7	5	0.669	0.754	0.875
8	6	0.621	0.707	0.834
9	7	0.582	0.666	0.798
10	8	0.549	0.632	0.765
11	9	0.521	0.602	0.735
12	10	0.497	0.576	0.708
13	11	0.476	0.553	0.684
14	12	0.458	0.532	0.661
15	13	0.441	0.514	0.641
16	14	0.426	0.497	0.623
17	15	0.412	0.482	0.606
18	16	0.400	0.468	0.590
19	17	0.389	0.456	0.575
20	18	0.378	0.444	0.561
21	19	0.369	0.433	0.549
22	20	0.360	0.423	0.537
23	21	0.352	0.413	0.526
24	22	0.344	0.404	0.515

A.4 Chi-square Distribution Table

df	Confidence Level								
	.995	.99	.975	.95	.9	.1	.05	.025	.01
1	0.00	0.00	0.00	0.00	0.02	2.71	3.84	5.02	6.63
2	0.01	0.02	0.05	0.10	0.21	4.61	5.99	7.38	9.21
3	0.07	0.11	0.22	0.35	0.58	6.25	7.81	9.35	11.34
4	0.21	0.30	0.48	0.71	1.06	7.78	9.49	11.14	13.28
5	0.41	0.55	0.83	1.15	1.61	9.24	11.07	12.83	15.09
6	0.68	0.87	1.24	1.64	2.20	10.64	12.59	14.45	16.81
7	0.99	1.24	1.69	2.17	2.83	12.02	14.07	16.01	18.48
8	1.34	1.65	2.18	2.73	3.49	13.36	15.51	17.53	20.09
9	1.73	2.09	2.70	3.33	4.17	14.68	16.92	19.02	21.67
10	2.16	2.56	3.25	3.94	4.87	15.99	18.31	20.48	23.21
11	2.60	3.05	3.82	4.57	5.58	17.28	19.68	21.92	24.72
12	3.07	3.57	4.40	5.23	6.30	18.55	21.03	23.34	26.22
13	3.57	4.11	5.01	5.89	7.04	19.81	22.36	24.74	27.69
14	4.07	4.66	5.63	6.57	7.79	21.06	23.68	26.12	29.14
15	4.60	5.23	6.26	7.26	8.55	22.31	25.00	27.49	30.58
16	5.14	5.81	6.91	7.96	9.31	23.54	26.30	28.85	32.00

References

[1] *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin Company Houston, 1990.

[2] *Core Java - Volume II - Advanced Features.* Rachel Bordeb, 2002.

[3] *Verification, validation and testing in software engineering.* IGI Global, 2006.

[4] *Software Testing Foundations*, chapter 2.1, pages 13–14. RockyNook, 2007.

[5] *JUnit in Action.* Manning Publications Co, 2011.

[6] 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*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

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

[8] J. Bach. Good enough quality: beyond the buzzword. *Computer*, 30(8):96–98, August 1997.

[9] B. W. Boehm. Guidelines for verifying and validating software requirements and design specifications. In P. A. Samet, editor, *Euro IFIP 79*, pages 711–719. North Holland, 1979.

[10] J.S. Bradbury, J.R. Cordy, and J. Dingel. Mutation operators for concurrent java (j2se 5.0). In *Mutation Analysis, 2006. Second Workshop on*, page 11, 2006.

[11] Jeff Carver, John Van Voorhis, and Victor Basili. Understanding the impact of assumptions on experimental validity. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 251–260, Washington, DC, USA, 2004. IEEE Computer Society.

[12] W. Robert Collins, Keith W. Miller, Bethany J. Spielman, and Phillip Wherry. How good is good enough?: an ethical analysis of software construction and use. *Commun. ACM*, 37:81–91, January 1994.

[13] F. Del Frate, P. Garg, A.P. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 124 –132, October 1995.

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

[15] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972.

[16] I. M. M. Duncan and D. J. Robson. Ordered mutation testing. *SIGSOFT Softw. Eng. Notes*, 15:29–30, April 1990.

[17] A. Eltaher. Towards good enough testing: A cognitive-oriented approach applied to infotainment systems. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 525–528, Washington, DC, USA, 2008. IEEE Computer Society.

[18] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 7 pp., april 2003.

[19] R. Finkbine. Usage of mutation testing as a measure of test suite robustness. In *Digital Avionics Systems Conference, 2003. DASC '03. The 22nd*, volume 1, pages 3.B.4 – 3.1–4 vol.1, 2003.

[20] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.

[21] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 55 –64, 2010.

[22] Don Gotterbarn. 'perfection' is not 'good enough': beyond software development. *ACM Inroads*, 1:8–9, December 2010.

[23] B.J.M. Grun, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pages 192 –199, 2009.

[24] R.G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, SE-3(4):279 – 290, july 1977.

[25] Almut Herzog and Nahid Shahmehri. Performance of the java security manager. *Computers and Security*, 24(3):192 – 207, 2005.

[26] Douglas Hoffman. Cost benefits analysis of test automation, 1999.

[27] W.E. Howden. Weak mutation testing and completeness of test sets. *Software Engineering, IEEE Transactions on*, SE-8(4):371 – 379, 1982.

[28] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 191 –200, May 1994.

[29] S.A. Irvine, T. Pavlinic, L. Trigg, J.G. Cleary, S. Inglis, and M. Utting. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 169 –175, sept. 2007.

[30] Y Jia and M Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 2010.

[31] Ying Jiang, Shan-Shan Hou, Jin-Hui Shan, Lu Zhang, and Bing Xie. Contract-based mutation for testing components. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 483 – 492, 2005.

[32] Cem Kaner, Copyright Cem, and Kaner All. The impossibility of complete testing, 1997.

[33] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 30 2010.

[34] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[35] Janusz Laski, Wojciech Szermer, and Piotr Luczycki. Dynamic mutation testing in integrated regression analysis. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 108–117, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[36] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Trans. Inf. Syst. Secur.*, 12:14:1–14:37, January 2009.

[37] Yu Seung Ma. Description of method-level mutation operators for java. Technical report, Electronics and Telecommunications Research Institute, Korea, 2005.

[38] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. *Software Reliability Engineering, International Symposium on*, 0:352, 2002.

[39] Yu-Seung Ma and Jeff Offutt. Description of class mutation mutation operators for java. Technical report, Electronics and Telecommunications Research Institute, Korea, 2005.

[40] Steve McConnell. *Code complete: a practical handbook of software construction.* Microsoft Press, Redmond, WA, USA, 1993.

[41] Steve McConnell. *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004.

[42] Sun Microsystems. java.security.accesscontrolexception, 2010.

[43] L.J. Morell. A theory of fault-based testing. *Software Engineering, IEEE Transactions on*, 16(8):844 –857, August 1990.

[44] Elfurjani S. Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[45] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 57–68, New York, NY, USA, 2009. ACM.

[46] A. Jefferson Offutt and W. Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, 1994.

[47] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5:99–118, April 1996.

[48] A. Jefferson Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. Technical report, George Mason University and Middle Tennessee State University, 2000.

[49] A.J. Offutt. A practical system for mutation testing: help for the common programmer. In *Test Conference, 1994. Proceedings., International*, 2 1994.

[50] A.J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pages 100 –107, May 1993.

[51] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. An experimental mutation system for java. *SIGSOFT Softw. Eng. Notes*, 29:1–4, September 2004.

[52] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. The class-level mutants of mujava. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 78–84, New York, NY, USA, 2006. ACM.

[53] Paul W. Oman and Curtis R. Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, CSC '90, pages 244–250, New York, NY, USA, 1990. ACM.

[54] Mike Papadakis, Nicos Malevris, and Maria Kallia. Towards automating the generation of mutation tests. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 111–118, New York, NY, USA, 2010. ACM.

[55] Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 85–91, New York, NY, USA, 2006. ACM.

[56] Drew Roselli and Thomas E. Anderson. Characteristics of file system workloads. Technical Report UCB/CSD-98-1029, EECS Department, University of California, Berkeley, 1998.

[57] Johanna Rothman. Determining your projectś quality priorities, 1999.

[58] D. Schuler and A. Zeller. (un-)covering equivalent mutants. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 45 –54, 2010.

[59] David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. In Gregg Rothermel and Laura K. Dillon, editors, *ISSTA*, pages 69–80. ACM, 2009.

[60] Lijun Shan and Hong Zhu. Testing software modelling tools using data mutation. In *Proceedings of the 2006 international workshop on Automation of software test*, AST '06, pages 43–49, New York, NY, USA, 2006. ACM.

[61] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 351–360, New York, NY, USA, 2008. ACM.

[62] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. *SIGPLAN Not.*, 44:37–48, March 2009.

[63] Dag I. K. SjÃ¸berg, Bente Anda, Erik Arisholm, Tore Dybå, Magne J"rgensen, Amela Karahasanovic, Espen F. Koren, and Marek Vokác. Conducting realistic experiments in software engineering. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, pages 17–, Washington, DC, USA, 2002. IEEE Computer Society.

[64] B.H. Smith and L. Williams. An empirical evaluation of the mujava mutation operators. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 193 –202, sept. 2007.

[65] M. Sridharan and A.S. Namin. Prioritizing mutation operators based on importance sampling. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 378 –387, nov. 2010.

[66] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. *SIGSOFT Softw. Eng. Notes*, 18:139–148, July 1993.

[67] Gerald M. Weinberg. *The psychology of computer programming (silver anniversary ed.)*. Dorset House Publishing Co., Inc., New York, NY, USA, 1998.

[68] J. Christopher Westland. The cost of errors in software development: evidence from industry. *J. Syst. Softw.*, 62:1–9, May 2002.

[69] Wikipedia. Blue screen of death, 2010.

[70] T.W. Williams, M.R. Mercer, J.P. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, 2001.

[71] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 230 –238, November 1994.

[72] M.R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 152 –158, July 1988.