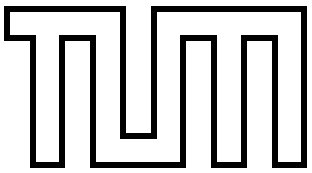


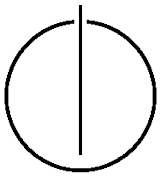
FAKULTÄT FÜR INFORMATIK

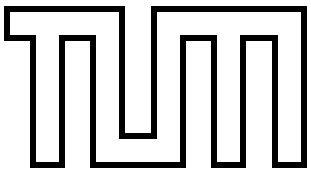
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Wirtschaftsinformatik

Meaningful and Practical Measures for Regression Test Reliability

Rainer Niedermayr





FAKULTÄT FÜR INFORMATIK

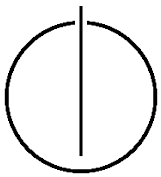
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Wirtschaftsinformatik

Meaningful and Practical Measures for Regression Test Reliability

Sinnvolle und anwendbare Messung der Zuverlässigkeit von Regressionstests

Bearbeiter:	Rainer Niedermayr
Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dr. Elmar Juergens Sebastian Eder
Abgabedatum:	September 13, 2013



I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, September 13, 2013 Rainer Niedermayr

Abstract

Successful software systems are subject to continuous maintenance and further development. Automated tests can help to detect faults caused by changes at an early stage and therefore play an important role in the software development process. In the practice, the code coverage metric is often used as a criterion to evaluate the reliability of test suites with focus on regression errors. However, the code coverage expresses only, which portion of a program has been executed by tests, but not, how reliable the tests actually are.
This thesis comprises on the one hand an analysis, how meaningful the code coverage is, and on the other hand the development of further criteria and measures to reveal inadequately tested code chunks. The measures were implemented in a prototype for Java applications based on the concept of mutation testing. It was a requirement that the execution of the analysis is applicable even to larger systems in reasonable time. Within a case study, 15 open source projects of different sizes with unit and system tests were considered and examined.
The results of the empirical evaluation indicate that the code coverage is acceptable as a rough approximation for the rating of the reliability of unit tests. However, the deviation of executed and actually tested code is too high in system tests and strongly depends on the software project, so that other criteria fit better for this type of tests.

Keywords: regression tests, test reliability, code coverage, mutation testing

Zusammenfassung

Erfolgreiche Softwaresysteme unterliegen einer kontinuierlichen Wartung und Weiterentwicklung. Automatisierte Tests können von Änderungen verursachte Fehler frühzeitig aufdecken und spielen aus diesem Grund im Softwareentwicklungsprozess eine wichtige Rolle. Zur Bewertung der Zuverlässigkeit von Testsuiten im Hinblick auf Regressionsfehler wird in der Praxis häufig die Testabdeckung (Code Coverage) als Kriterium verwendet. Allerdings gibt die Code Coverage nur Aufschluss darüber, welcher Anteil des Programms von Testfällen durchlaufen wird, nicht aber darüber, wie zuverlässig wirklich getestet wird.
Im Rahmen dieser Masterarbeit wurde zum einen untersucht, welche Aussagekraft die Code Coverage aufweist, und zum anderen wurden weitere Kriterien und Messungen zur Aufdeckung scheingetesteter Codestellen entwickelt. Die Messungen wurden in einem Prototyp für Java-Anwendungen nach dem Ansatz des Mutation Testings umgesetzt. Dabei wurde berücksichtigt, dass die Ausführung auch für größere Softwaresysteme in angemessener Zeit möglich ist. Im Rahmen einer Fallstudie wurden mit dem Prototypen 15 unterschiedlich große Open-Source-Projekte mit Unit- und Systemtests untersucht und anschließend ausgewertet.
Der empirischen Evaluation zufolge eignet sich die Code Coverage als grobe Annäherung zur Bewertung der Zuverlässigkeit von Unit-Tests. Bei den Systemtests ist die Abweichung zwischen durchlaufenem und tatsächlich getestetem Code allerdings zu hoch und sehr projektspezifisch, sodass hierfür andere Kriterien besser geeignet sind.

Schlüsselwörter: Regressionstests, Testzuverlässigkeit, Code Coverage, Mutation Testing

Contents

List of Figures

List of Tables

Listings

1 Introduction

Automated tests are used in many software projects to ensure the reliability of the software. During the development of the software, some code might get broken unintentionally. These tests can reveal faults in code chunks which were working properly at an earlier point of time. Such regression tests help to discover faults before their repair gets more expensive or the software is shipped to the user.
An important question is, how good the tests are at revealing faults. Information about how likely it is, that a test suite detects broken code, helps to better allocate quality assurance efforts.
In practice, the code coverage is often used as metric to judge the test effectiveness. However, the code coverage measures only which proportion of the whole code was executed by the tests. It does not express, how much actually got tested.
The following code example illustrates this weakness of code coverage as measurement:
The class `Calculation` consists of an internal field and of two public accessible methods. The field is named `value` and is of the type integer. It is initialized to 0 and describes the current state of the instance. The method `add(int x)` allows adding the value of `x` to the current state. The method `isEven()` returns a boolean value, which indicates whether the current state represents an odd or an even number.

```
public class Calculation {
  private int value;

  public Calculation() {
    this.value = 0;
  }

  public void add(int x) {
    this.value += x;
  }

  public boolean isEven() {
    return this.value % 2 == 0;
  }
}
```

Listing 1.1: Class under test

The class `Calculation` is tested by `CalculationTests`, which consists of two JUnit testcases: `testIsEven1` creates a new instance of `Calculation` and checks, whether the state is even after the instantiation. This applies, since the state is set to 0 at the beginning. `testIsEven2` also creates a new instance, adds the value 6 and verifies that the state is still even. This assertion is appropriate, too.

```
public class CalculationTests {
  @Test
  public void testIsEven1() {
    Calculation c = new Calculation();
    assertTrue(c.isEven());
  }

  @Test
  public void testIsEven2() {
    Calculation c = new Calculation();
    c.add(6);
    assertTrue(c.isEven());
  }
}
```

Listing 1.2: Class with testcases

The code coverage of the class under test is 100% since each method was executed by at least one testcase. But does that say that the class `Calculation` has been perfectly tested?

Not really. Consider the `add` method, which was executed only by the second testcase. If the programmer had forgotten to implement the body of the method (and its logic had been empty as a consequence), would the testcase have detected that fault? No, it would not. The state of `value` would not have changed to 6, but since 0 and 6 are both even numbers, the assertion would not have caused the testcase to fail.
Thus, the coverage pretends a false sense of correctness. All three methods were executed, but only two of them (66.7%) were actually tested.
Testcases without any assertions are another example of tests, which increase the coverage, but are mostly useless.[1]

Goal

The first goal of this thesis is to determine, how reliable code coverage is. It attempts to answer the question of how much of the covered code is actually tested. Even though the code coverage measure does not analyze the effectiveness of a test suite in depth, it might still be suitable as an approximation for the result.
The second goal is to find other metrics to express the reliability of automated testcases. Characteristics of inadequately tested methods, characteristics of weak testcases and the relationship among them can compose such metrics. Their advantage is that they can be evaluated by using static program analysis rather than having to execute all the testcases. It makes them applicable to huge software systems, too.

Thesis Outline

This thesis is structured as follows:
Chapter 2 introduces the used terminology and gives an overview of the concepts of code coverage and mutation testing.
Chapter 3 sheds light on related work by presenting and discussing existing approaches and by describing, how this work differs from and extends existing work.
Chapter 4 presents the approach taken in this work. It describes the steps of the workflow and the asymptotic runtime complexity.
Chapter 5 is about the prototypical Java implementation of the approach described in the previous chapter. It covers the architecture, realization details and Java specific peculiarities.
Chapter 6 contains a case study, which uses the prototypical implementation to collect data from real world software projects. The obtained data is evaluated and the results are presented, analyzed and discussed.
Chapter 7 covers open questions which need further research. Furthermore, possible extensions to the approach are presented.
Finally, chapter 8 summarizes this thesis and gives a conclusion.

1 Nevertheless, Martin Fowler writes that “although assertion-free testing is mostly a joke, it isn’t entirely useless”. He was reminded that “some faults [such as null pointer exceptions] do show up through code execution”. [1]

2 Fundamentals

This chapter introduces the used terminology and gives an overview of the concepts of code coverage and mutation testing.

2.1 Terminology

2.1.1 Testing

The following terminology is related to testing.

Testcase

A *testcase* executes a certain part of a program and compares the computed result with the expected one. Automated testcases are often realized as methods and executed with the use of testing frameworks such as JUnit or TestNG. Depending on the extent and focus of a testcase, a differentiation in unit and system tests is possible.

Unit Test

A *unit test* tests a small unit of code (a method or a class) intensively. It consists of a sequence of method calls and assertions checking that the computed results of the invocations equal the expected ones. It is also possible to check the absence of thrown exceptions for a given program flow.

System Test

A *system test* evaluates an integrated system and focuses on the interactions among components. Unlike a unit test, it covers a lot of methods by executing a large proportion of the whole system. A system test often triggers the execution of a large workflow and compares the end result (which can be aggregated data, a report, a log file, etc.) with the expected one.

Regression Test

All automated tests (including unit and system tests) which are executed periodically with the purpose of revealing regression errors are considered as *regression tests* in this thesis.

Test Class

A class is considered as *test class* if it contains at least one testcase. It may contain testcases directly or inherit them. Besides testcases, a class may also contain set up, tear down and helper methods.

Program Class

A *program class* contains program logic and is used during the normal execution of a program. It is the object of investigation of tests.

Test Suite

A *test suite* bundles several test classes and allows executing all their testcases in once. It is a way to group testcases. A test suite may be represented as a test class.

Testing Framework

A *testing framework* supports the creation and execution of testcases. The framework may provide utilities to verify computed data, may allow specifying procedures, which are triggered before or after the execution of testcases, may allow specifying rules (e.g. timeouts), etc.
JUnit and TestNG are popular frameworks.

Regression Test Reliability

The *regression test reliability* expresses the capability of a test suite to detect newly occurring errors in existing code.

Error

According to ISO 26262, an *error* is a "discrepancy between a computed, observed or measured value or condition, and the true, specified, or theoretically correct value or condition". [2]

Failure

According to ISO 26262, a *failure* is the "termination of the ability of an element, to perform a function as required". [2]

Fault

According to ISO 26262, a *fault* is an "abnormal condition that can cause an element or an item to fail". [2]

2.1.2 Mutation Testing

The terminology related to mutation testing is explained in chapter 2.3. In addition to the terminology used in the literature, the following terms are introduced:

Mutation Testing Result (of a Method)

The *mutation testing result* is the analysis result of a method for which mutants were generated and executed against at least one testcase. It can either be *not inadequately tested* if at least one of the method's mutants was killed by any testcase or *inadequately tested* if no single mutant was killed by any testcase.

Method Mutation Score

The *method mutation score* is an aggregated measure and expresses the ratio of the not inadequately tested methods out of all analyzed methods of a project. The score ranges from 0 to 1.
A method mutation score of 80% expresses that 80% of the analyzed methods are not inadequately tested.

Test Score

The *test score* expresses the ratio of methods a testcase detects as faulty (due to the mutation) out of the methods it executes. The score ranges from 0 to 1.
A testcase which executes 20 different methods and detects 5 out of these as faulty has a test score of 25%.

2.1.3 Java

The following terminology is related to Java.

Bytecode

Java compilers translate the source code of classes into *bytecode*. The bytecode is an intermediary representation of Java programs and executed by the Java Virtual Machine (JVM). [3]
The code describes the structure of the classes and contains a sequence of instructions for each method. Each instruction corresponds to an operation code (opcode) and is followed by zero or more operands. [4]
It is possible to analyze, generate or transform classes at runtime before they are loaded into the Java Virtual Machine (JVM). The advantages of working on the bytecode are that it can be modified way faster than the source code (which needs recompiling) and that the source code does not need to be available.

Class Loader

The *class loader* is in charge of loading a class (denominated by its package and class name), when it is needed the first time during the program execution. The Java standard class loader first seeks the class in the Java Runtime Environment (JRE). If the class is not part of the JRE, the class loader will go through the artifacts specified in the classpath and pick the first occurrence of the compiled class.
The standard class loader can be overwritten to achieve another behavior.

Classpath

The *classpath* is a string listing paths of bytecode artifacts, which contain classes needed during the execution of a program. The artifacts can either be jar files or compiled classes in a folder structure representing the packages.
As the class loader goes through the listed entries and picks the first matching class when seeking one, their order on the classpath is relevant. This needs to be considered if two different versions of a class are on the classpath.

Factory Pattern

The *factory* is a design pattern. It describes a class which is responsible for the instantiation of objects.

Generics

Java supports the concept of *generics* since version 5. Generics allow designing data structures in a more general way without losing type safety. When declaring the data type of a certain element, placeholders can be used instead of concrete types. The concrete type is specified when using the class.
For instance, lists are implemented with generics in Java. The allowed type for the list elements is specified, when the list is instantiated.

Overloaded Methods

Overloaded methods are methods which are contained in the same class and have the same name but differ in the number, order or type of the parameters.

2.2 Static and Dynamic Code Analysis

The purpose of source code analysis is to collect information about the program code.
Binkley defines it in [5] as "the process of extracting information about a program from its source code or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools". It can be performed at edit, compile, link or run time.

According to Binkley, the source code analysis consists of three components:
First, a parser converts the code into an internal representation. Most parsers are compiler-based.
Second, the internal representation is usually transformed to focus on a particular aspect of a program and to be better suited for the particular analysis. It can also be combined with the results of other analyses. Examples for internal representations are control-flow graphs, call graphs, abstract syntax trees, etc.
Third, the actual analysis is performed. Algorithms examine the internal representation to compute the result.

A classification into static and dynamic analysis is possible:
The static analysis is performed on the source code or on the bytecode. It does not take into account the program input. Thus, the gained results apply to all executions of the program.
The dynamic analysis considers the program input. This type of analysis includes the execution of some parts of the program. It allows a greater precision, however, it is usually more time consuming and the results depend on the particular input.

Applications of the source code analysis include compilers, debuggers, profilers, clone detection, reverse engineering, etc.

[5]

2.3 Mutation Testing

Mutation testing is an established technique to evaluate test suites. It was first proposed in the 1970s. The general principle is to introduce faults into a program and to check, whether the tests can detect these. A program is considered as well tested, if most introduced faults can be detected.

The traditional mutation testing targets on faults which are close to the correct version of the program. These simple faults are assumed to simulate all possible faults because of the following two hypotheses:
The *Competent Programmer Hypothesis* (CPH) states that programmers are competent and hence develop programs close to the correct version. Consequently, most faults are only small syntactic deviations which lead to an incorrect program behavior. The CPH was introduced by DeMillo *et al.* in 1978.
The second hypothesis is the *Coupling Effect* and was also proposed by DeMillo *et al.* It states that complex faults arise from the composition of simple faults and that tests which are able to detect simple faults are sensitive enough to detect complex ones, too.
[6] [7] [8]

The mutation analysis consists of two steps:
The first step involves the creation of faulty versions of a program. The process of seeding faults is called *mutation* and the resulting code artifact is called *mutant*. Most mutation tools create mutants which contain a single fault (*single-order mutants*). Though, it is also possible to create so called *higher-order mutants* which contain multiple changes. The transformation rule, which describes how the code is altered to introduce faults, is called *mutation operator*. Table 2.1 lists a selection of mutation operators.

Operator	Description
AAR	array reference for array reference replacement
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
CRP	constant replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	return statement replacement
SDL	statement deletion
SVR	scalar variable replacement

Table 2.1: Some mutation operators (excerpt from [6])

In the second step, the mutants are executed against the tests. A mutant is said to be *killed*, if at least one test can detect the fault (i.e. is able to detect that the program does not behave as it should). Otherwise, a mutant is called *living*.

Equivalent mutants are mutants which differ syntactically from the original code, but are semantically equal. They cannot be killed by any test and distort the results by indicating weaknesses in the test suite which actually do not exist. Therefore, they need to be removed from the results. However, it is not possible to automatically detect all equivalent mutants, because the equivalence of programs is not decidable in general. The detection is an instance of the infeasible path problem. [9]

The *mutation score* is the aggregated result of the mutation analysis. It indicates the quality of a test suite and corresponds to the percentage of the non equivalent mutants killed. A test suite is *mutation adequate*, if its mutation score is 100%.

All in all, mutation testing is a powerful technique. Experimental studies ([10], [11], [12]) provide evidence that it is a good indicator for the fault detection ability of test suites.
Besides test suite evaluation, mutation testing can also be used to support other testing activities:

- It can assist the automated testcase prioritization. Testing sequences are scheduled according to their rate of killed mutants.
- Mutation testing can be used to minimize tests sets. Tests which cover only mutants that were already killed by other tests can be excluded, because they do not add value.
- Mutation testing renders the generation test data that can effectively kill mutants possible. This is known as *constraint-based test data generation* (CBT).

[6] [13]

However, mutation testing has two major problems: The computational costs are high because the analysis involves the creation of a high number of mutants and the execution of each one against the test suite. The second problem is the undecidability of the equivalent mutants.

Chapter 3.2 presents related work to mutation testing. A considerable portion of the related work addresses these two problems and discusses approaches to overcome them.

3 Related Work

This chapter sheds light on related work by presenting and discussing existing approaches and by describing, how this work differs from and extends existing work.

3.1 Code Coverage

The code coverage expresses to what degree the tests cover the code of a program. The general idea is that tests can find only failures in code they execute.

Different code coverage criteria exist:

- The statement coverage (C_0) expresses which proportion of the statements is executed by tests.
- The decision coverage (C_1) considers which proportion of branches in methods is covered by tests. It is also known as branch coverage.
- The condition coverage (C_2 / C_3) checks the evaluation of conditions. Each condition is supposed to be evaluated at least once to true and once to false by tests.
- The path coverage (C_4) examines the control flow. It gives information about the proportion of covered paths. Since it is not feasible to test all theoretical possible execution paths (especially loops lead to a huge number of paths), alleviated variations of this criterion exist.

[14] [15]

Popular Java tools to compute different code coverage metrics are *EclEmma* and *Clover*.

The code coverage is often used as a quality metric for the tests. However, it has some flaws. First, the coverage expresses only that certain code was executed by tests. It does not consider, if the code is likely to contain faults. Second, the execution of statements or branches does not imply that it is performed with all possible values. Third, the tests might not be able to detect faults, because they are not performing (enough) checks when executing the code or do not recognize the output as faulty. [16] [17]

This thesis tries to answer, whether the coverage can be used as an approximation for the reliability of a test suite. Furthermore, it presents a mutation testing approach, which examines the test reliability in a deeper analysis.

3.2 Mutation Testing

Related work in the area of mutation testing primarily addresses mutation operators, optimizations to reduce the execution costs and the problem of equivalent mutants.

3.2.1 Mutation Operators

Several papers present and discuss new sets of mutation operators which target at specific fields of application and complement the traditional operators.

Derezińska presents mutation operators which regard special characteristics of object oriented software in [18]. The operators concern features of the program that "directly refer to the construction and control of classes and objects".
Derezińska classifies the object oriented mutation operators into five groups:

- inheritance: replace the super class, change the direction of the inheritance relation or remove it

- association: change an association between classes
- object: call a method on a different object of the same class or of another class in the hierarchy
- member: call another method of the class than the specified one
- access: change an access modifier to a more restrictive one

The faults represent the misuse of classes and object interrelations. The author shows that the mutations can also be performed on UML class diagrams.

Alexander *et al.* also address the object oriented mutation in [19]. They focus on Java programs and provide an approach to mutate classes which implement certain interfaces. Their mutation engine works on methods of collections, maps, iterators, etc. One operator creates for example mutants in which iterators skip a certain number of elements when iterating over a collection.

Offutt and Praphamontripong present a novel solution to the problem of integration testing of web applications in [20]. They define new mutation operators which concern web applications and can be applied on the HTML code of java server pages (JSPs) as well as on the corresponding servlets.

Mateo *et al.* provide mutation operators suited for composed software systems in [21]. The operators alter configuration files, interchange components of systems with another versions, remove graphical components from the user interface, etc.

Traditional and object-oriented operators are insufficient at finding faults which cause deadlocks, starvations and related problems in concurrent code. Therefore, Bradbury *et al.* introduce additional mutation operators for concurrent Java programs ([22]). Their operators focus at the mutation of code chunks which are responsible for synchronization aspects.

3.2.2 Reduction of Execution Costs

Several techniques were proposed to reduce the execution costs of mutation testing. Offutt and Untch categorize these techniques into three strategies: *do fewer*, *do smarter* and *do faster* [23]

Do Fewer

The *do fewer* strategy tries to create less mutants without incurring intolerable loss in effectiveness.

The *selective mutation* is one approach of this strategy. The idea is to use only the most critical mutation operators since some operators add only little effectiveness. Offutt *et al.* indicate that 5 out of the 22 operators used by the Mothra tool[1] are sufficiently effective ([24]). Mresa and Bottaci conduct an empirical study on different selective mutation strategies and present their results in [25]. They state that the use of a reduced set of operators provides significant efficiency gains and can be applied if the mutation score is not required to be very close to one. Their results suggest that a strategy which randomly selects which generated mutants are executed against tests can achieve a higher mutation score.

Another "do fewer" approach is to create mutants of *higher order*. Second-order mutants contain two faults by combining two first-order mutants. Thereby, the number of mutants can be reduced to half. Furthermore, a mutant which contains two faults is less likely to be an equivalent mutant. Polo *et al.* conducted an experiment and came to the conclusion that "second-order mutation is adequate to estimate the quality of test cases". The risk exists that faults remain undiscovered, but it is acceptable for second-order mutants. They state that even kth-order mutation (with $k > 2$) can be accepted as cost-effective testing practice for non-critical software. The choice of k is a tradeoff between the costs and the accepted risk. [26]

1 Mothra is a mutation testing tool for Fortran applications. It was published in 1987.

Do Smarter

The goal of the *do smarter* strategy is to avoid unnecessary parts of the execution and to distribute the computation over several machines.

The *weak mutation* belongs to this strategy. It is an "approximation technique that compares the internal states of the mutant and original program immediately after execution of the mutated portion of the program". Thus, the tests are not necessarily executed until the end. Empirical results indicate that about 50% of the execution time can be saved with this technique without losing much test effectiveness. [23] [27]

Other "do smarter" approaches include the parallelization of the execution on multiple machines and algorithms which intelligently cache reusable state information. [23]

Do Faster

The *do faster* strategy covers approaches that speed up the generation and execution of mutants.

Untch developed the *Mutant Schema Generation* method ([28]). Instead of generating many mutants for a certain code chunk, this method generates one "metamutant" which represents all possible faults. Before running the tests, a flag is set to select the desired fault. This method, which reduces the effort needed to generate mutants, is especially useful if the mutation process involves the costly compilation. [29]

Another way to speed up mutation testing in some programming languages is to use *bytecode transformations* when creating mutants. It saves the compilation costs and allows testing projects for which the source code is not available. [6] [30]

3.2.3 Equivalent Mutants

Grün *et al.* investigate in [31] the impact of equivalent mutants. They give evidence that equivalent mutants are quite common and describe a conducted experiment in which 8 of 20 mutants turned out to be equivalent.
Their main finding is that the impact of a mutation on the program execution can be used to identify equivalent mutants. The less a mutation alters the execution, the higher is the chance of being equivalent. The impact on the program execution can be determined for example by comparing the control flow before and after the mutation.
Moreover, Grün *et al.* classify equivalent mutants into four groups:

- Some mutants are equivalent because they concern unneeded code which does not affect the program. This is the case, if parts of the code duplicate some behavior or if values are overwritten before they are used.
- Some equivalent mutants suppress speed improvements (e.g. a key-value-pair is put multiple times into a map due to a mutation).
- Some equivalent mutants alter the private state of a class or the return value of private methods without affecting the behavior.
- Some equivalent mutants cannot be triggered because conditions are not met.

Baldwin and Sayward suggest in [32] to use compiler optimization strategies for the detection of equivalent mutants. The general idea is that some of the equivalent mutants are optimizations or de-optimizations of the original program. Offutt and Craft take up this idea and implement the algorithms in an experimental tool. They are able to detect a significant percentage of the equivalent mutants. [33]

3.2.4 Differentiation

The mutation testing approach applied within the context of this thesis is primarily a mean to collect mutation results of software projects. The focus is on the evaluation of the obtained results with the goal to determine the reliability of code coverage and to find other indicators for inadequately tested methods.

Concerning the mutation operator, a single one is used which mutates the whole body of a method. It is supposed to reveal methods which are tested in a really weak manner.
Equivalent mutants are not of a major concern. As the mutation operator is considered to have a high impact on the program execution, the number of equivalent mutants should presumably be low. Nevertheless, identifiable ones are filtered out. (The used operator is a special case which permits identifying a high ratio of equivalent mutants.)
The execution costs of the approach are relevant. A reasonable runtime is achieved through parallelization (multiple concurrent processes), mutation on bytecode level and selective mutation (use of a single mutation operator).

3.3 Mutation Testing Tools

A couple of mutation testing tools exist for Java programs. Popular tools are *PIT Mutation Testing*, *Javalanche*, *MuJava*, *Judy* and *Jester*. They differ in terms of mutation operators, efficiency and provided features.
The applicability, computed analysis results and runtimes of all five tools are examined and compared in the case study (see research question 3.1 in chapter 6.5.3).

3.3.1 PIT Mutation Testing

Many mutation testing tools for Java are slow, difficult to use and written to meet the needs of academic research rather than the ones of development teams in practice. *PIT Mutation Testing* is a relatively new tool and aims to be different. The author considers it to be fast, scalable and suitable for real world software.
The tool is also available as Eclipse plugin. It provides 10 mutation operators, out of which 7 are enabled by default. The mutation is performed on the bytecode.
PIT contains an *incremental analysis* as experimental feature: It can track changes to the code base and decide from this information, which methods need to be mutated again in the next run in the future. It bases on the not yet proven assumption that changes in the dependencies of a class seldom change the result of a mutation test. This feature allows reusing the results of previous analyses and consequently most unchanged code chunks do not need to be retested (under the condition that the corresponding tests were not altered). It greatly reduces the computational costs of subsequent runs. [34]

3.3.2 Javalanche

Javalanche is an open source framework for mutation testing developed by the Saarland University. Javalanche provides a special feature called *impact detection*, which analyzes the impact of a mutation on the behavior of the program. The greater the impact on the behavior is, the less likely it is that the mutant is equivalent. This technique shall help to filter out equivalent mutants quite well.
In order to reduce the number of mutants, the program applies only a small but sufficiently complete set of possible mutation operators *(selective mutation)*.
Further optimizations include the direct manipulation of the Java bytecode, the parallel execution of the mutants and the filtering of tests which do not cover mutated statements. [30]

3.3.3 MuJava

MuJava was first released in 2003 by Jeff Offutt. The current version is MuJava 4 and was released in 2013. It has a graphical user interface and is the first version to support JUnit tests. (Previously, the tests were supposed to be methods returning a string describing the state of an object. In order to determine whether a mutant was killed, MuJava compared the string before and after the mutation. [21])
MuJava supports 15 mutation operators on method level (which mutate instructions within a method) and 28 operators on class level (which try to imitate object orientated faults) in the current version.
The program contains two separated GUIs:

In the first, the user can choose a class for mutation and select the desired mutation operators. Then, the program generates one source file per operator and compiles it (for this reason the source code is needed as input). Depending on the size of the class and the number of selected operators, the generation might take a while. If all operators are chosen, a huge number of mutants is generated.
After having generated mutants, the user can select a testcase in the second GUI and execute all mutants against it.
MuJava is not fully automated and its primary use is to analyze single test classes rather than whole programs. No features are provided to reduce the number of equivalent mutants. However, it provides a lot of mutation operators and extensions for special purposes (e.g. testing web applications).
[20] [29] [35]

3.3.4 Judy

Judy is another mutation testing tool for Java. It was developed by the Wroclaw University of Technology. The latest stable release is version 2.0 from 2011.
Judy performs the mutations on bytecode level and facilitates distributed and parallel computations over multiple machines. A special feature is the support of *higher order mutations.* [36]

3.3.5 Jester

Jester is a further mutation testing tool. It performs the mutations on source code level. Versions exist also for Python and C# program code. According to the author [37], "Jester's mutations are very simple text search and replace". Thus, it may happen that Jester "mutates in such a way that it no longer compiles". This is a problem, because a known and unresolved issue is that Jester will hang up, if a code chunk cannot be compiled or a dependency is missing. Jester seems to be out of development.

4 Analysis Approach

This chapter presents the approach taken in this work. It describes the steps of the workflow and the asymptotic runtime complexity.

4.1 General Idea

The general idea is to apply a mutation testing approach in order to analyze and evaluate test suites of projects. The whole workflow consists of four steps. As a preparation, the source code is instrumented in the first step. After that, information about testcases and methods under test is collected in the second step. These first two steps are executed once for the analysis of a project. Once they are completed, all the necessary information is available to proceed to the analysis steps. In the third step, one method under test at once is mutated and consequently tested in the last step. The last two steps are executed for each method under test and can be executed concurrently.

The state chart in figure 4.1 gives an overview of the workflow.

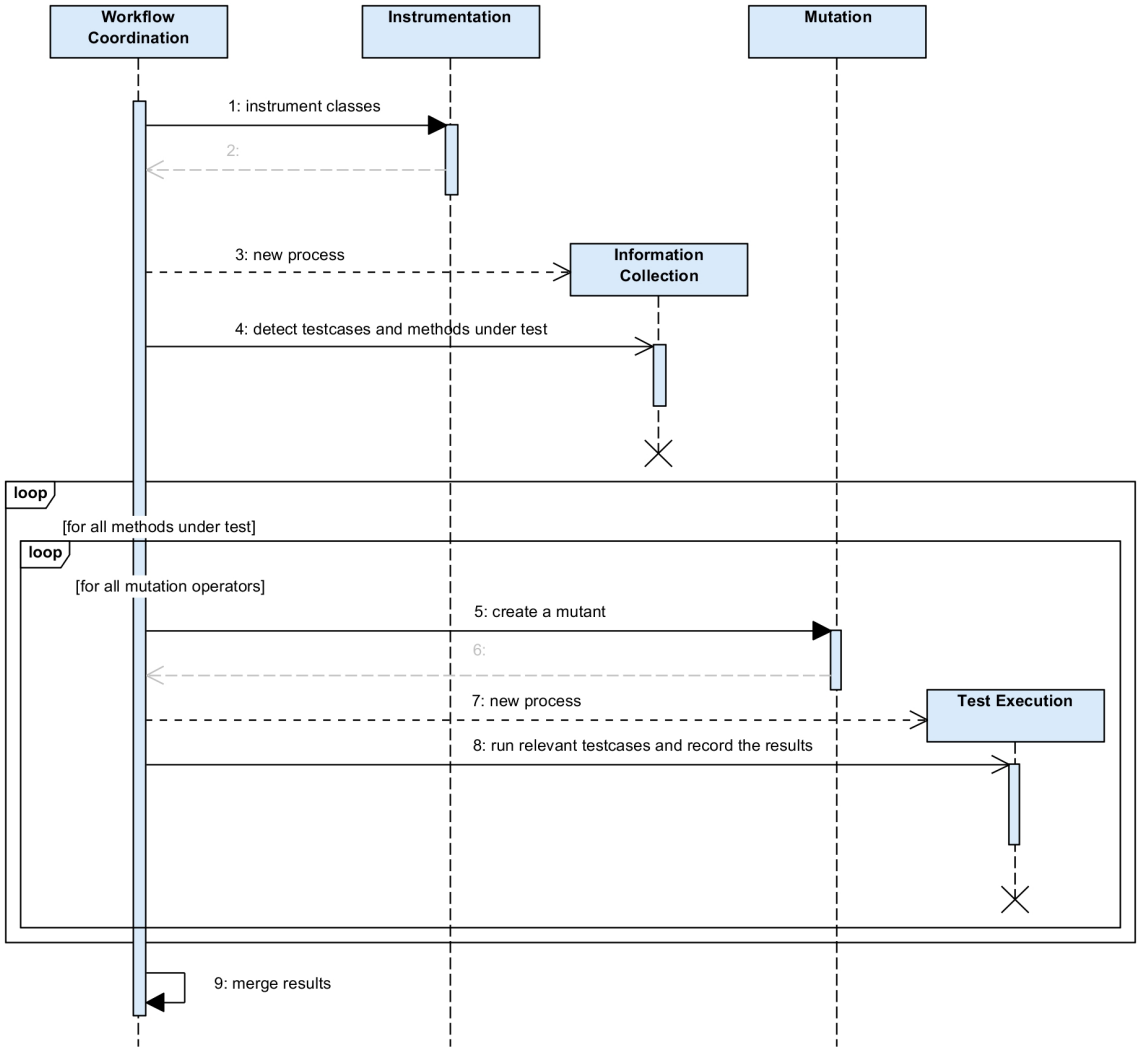


Figure 4.1: Overview of the whole workflow

The following sections describe the steps in detail.

4.2 Instrumentation

The *instrumentation step* takes the project to be analyzed as input. The project may consist of many artifacts. The instrumentation logic iterates over all contained class files and determines, whether a class is a program class or a test class. The determination is performed by analyzing the class hierarchy, its methods and its annotations. The concrete procedure depends on the testing framework of the analyzed project and is described in the implementation chapter.

4.2.1 Instrumentation of Program Classes

All classes which are not test classes are considered as program classes. The instrumentation procedure first retrieves all methods of the class. Then, special methods such as constructors and static initializers are filtered out. They are not relevant for the intended mutation operation.
Subsequently, an identifier is generated for each method, which allows identifying it uniquely. The identifier consists of the name of the package, the name of the class (including potential outer classes), the name of the method and the ordered list of the argument types.
Next, the code of the method is modified: At the beginning of the method logic, a statement is inserted. The statement causes the invocation of a logger with the method identifier as parameter value. The logger records all invocations.

4.2.2 Instrumentation of Test Classes

Test classes are instrumented differently. For each class, all testcases are retrieved. Then, the code of the testcases is modified: Before all existing instructions a new one is inserted. It indicates the logger class to get active. After the execution of the testcase, another statement which causes the logger to get inactive is added. It is inserted in a way that ensures the execution of the statement for all possible execution paths.
The instrumentation of testcases is necessary for the logger in order to capture only methods under test which are executed by the actual testcase. It is a way to distinguish the framing of the testcases, i.e. the set up and tear down procedures in other methods (which are automatically triggered by the testing framework), from the actual testing logic. Only the testcase itself is supposed to contain assertions and thus, only it should try to kill mutants.

4.3 Information Collection

After the instrumentation of the program and test classes, the *collecting of information* can be started. The purpose of this step is to gain information about the relationship between testcases and methods (compare figure 4.2).

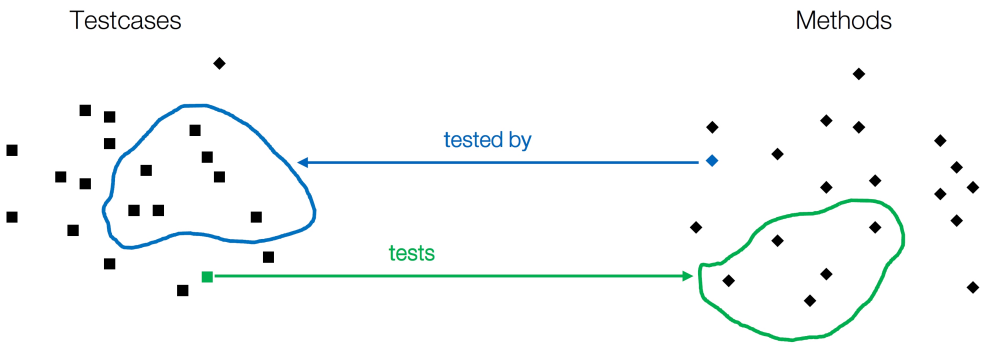


Figure 4.2: Relationship between testcases and methods

The execution of this step begins by retrieving again all classes of the analyzed program and determining their type (program or test class). Then, an iteration is started over all testcases of all test classes.
For each testcase, the following is performed: First, the logger, which is in charge of recording the method invocations, is reset. Then, the testcase is executed using the API of the appropriate testing framework. The framework executes set up procedures, runs the testcase itself and triggers the tear down. When executing the testcase, the logger is set active by the first statement of the testcase and switched off after the completion of the testing logic.
The execution ends with a result: If no exception is thrown during the execution and all assertions are satisfied, the testcase is successful. In the other case, it fails. A testcase which fails on the originally (not yet semantically modified) code is useless and will be skipped from this point on. The logged data of a successful testcase is retrieved. It corresponds to an unordered set, which contains all methods which were invoked during the actual testing logic. The data is persisted and will be used by the following step.
The state chart in figure 4.3 illustrates the procedure. (To keep it simple, the testcase invokes only one method and is not preceded respectively followed by any set up or tear down method.)

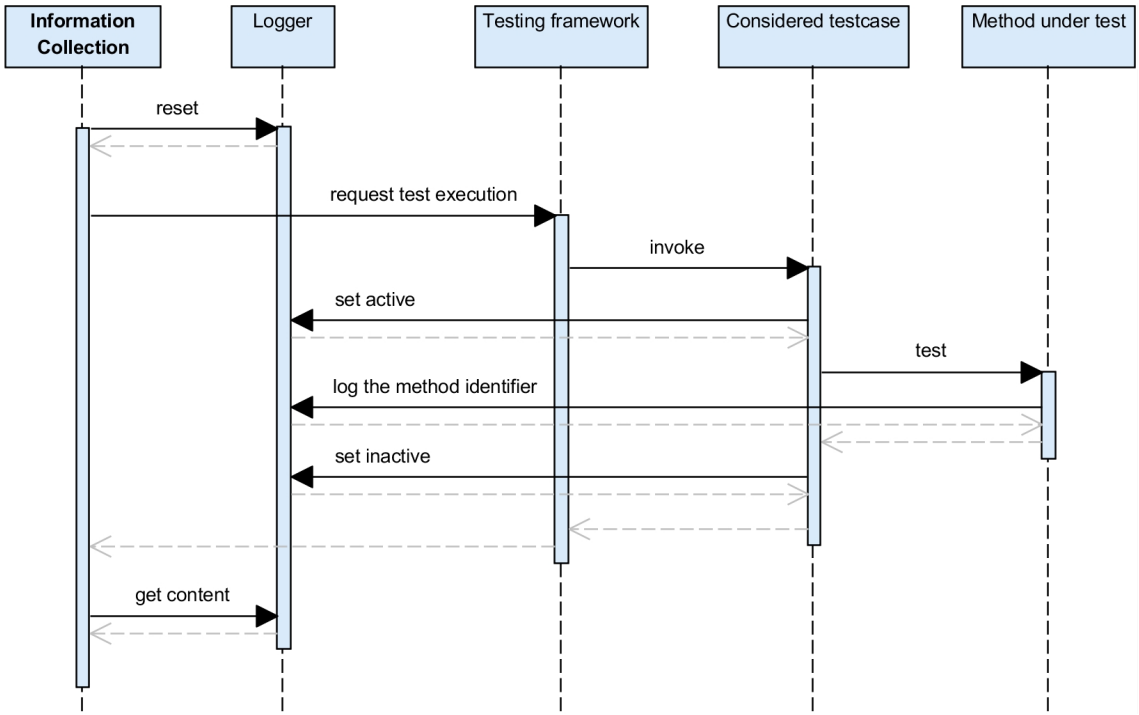


Figure 4.3: State chart illustrating the process to collect the necessary information

After all testcases have been processed, the collected data states for all testcases, which methods are directly or indirectly (i.e. through another method) invoked by a certain testcase. Furthermore, it is possible to calculate the other direction of the relationship: This is all the testcases which cover a certain method. It is an important information for the following execution of the mutants.

4.4 Mutation

The *mutation step* is executed for each of the previously collected methods; all of them are covered by at least one testcase. The step takes a class and the identifier of one of its methods as input and generates a mutant, i.e. a modified copy of the class in which the logic of the specified method differs from its original one.
The procedure for each of the methods is the following:

First, it is analyzed whether the mutation of the method is feasible and desired. The feasibility

Return type	Mutant 1	Mutant 2
boolean	`false`	`true`
byte, short, int, long	`0`	`1`
float, double	`0.0`	`1.0`
char	`' '`	`'A'`
string	`""`	`"A"`

Table 4.1: Values generated for primitive types

depends on the return type of the method and might not always be given (see later). Some methods are not desired, because they create equivalent mutants or are too trivial. Therefore, they can optionally be excluded from the analysis. They are:

- methods which contain no statements (empty methods)
- single-line setters which simply assign a single parameter value to a field (without any check or transformation)
- single-line getters which simply return the value of a field (without any transformation or branching)
- single-line getters which always return the same constant value

Furthermore, constructors and static initializers are not part of the analysis.

If the analysis shows that the mutation of the method is feasible and desired, it will be performed on the bytecode of the class as specified:
The mutation operator removes all statements and try-catch blocks. Subsequently, a return value is generated. The generation depends on the declared return type of the method:

- For void methods, no return value is needed.
- For methods which return a primitive value or a string, a constant is used. Table 4.1 lists these constants.
  (For these methods two mutants with different return values are generated.)
- For methods which return an object, a factory is used to generate an instance. The factory can consult, apart from the declared return type, the name of the method for the selection of an appropriate instance, too.
  If the factory cannot provide a suitable instance, the mutation will not be performed. The previous analysis will have determined the infeasibility.
  (Another possible option would be to generally return null. However, this causes many null pointer exceptions.)

Then, the value is put onto the stack (except for void methods). The insertion of a suitable return statement completes the mutation. Finally, the modified class is persisted.

As a consequence of the mutation, void methods will behave as if they are not invoked. In this case, the mutants represent faults caused by missed methods calls. Non-void methods additionally cover the use of a wrong return value.

The mutation operator is quite severe since its change has a significant influence on the behavior. This applies especially to longer methods. Thus, the mutation should reveal methods which are tested in a really weak manner.

4.5 Test Execution

The step *test execution* is executed exactly once for each mutant. It is supplied with the mutant and receives information about all testcases covering the corresponding method as input.
Its goal is to execute the mutant against all testcases. This is done using the testing framework. The result of each test run, i.e. whether the testcase succeeded or failed (and with what failure), is recorded along with information about the method identifier and the testcase name.

One point to be considered when running tests is to specify a maximum allowed time for the execution. This is especially important in mutation testing, because mutants might cause the

```
public List<Integer> compute() {
  List<Integer> list = new LinkedList<
      Integer>();
  list.addAll(computeInternal());
  return list;
}
```

⇓

```
// Method descriptor ()Ljava/util/List;
// Stack: 2, Locals: 2
public java.util.List compute();

new java.util.LinkedList
dup
invokespecial java.util.LinkedList()
astore_1
aload_1
aload_0
invokespecial de.tum.in.ma.example.
    Calculation.computeInternal()
invokeinterface java.util.List.addAll(
    java.util.Collection)
pop
aload_1
areturn
```

⇒

≡

```
public List<Integer> compute() {
  return (List) CalculationFactory.
      INSTANCE.get("de.tum.in.ma.example
      .Calculation.compute()", "java.
      util.List");
}
```

```
// Method descriptor ()Ljava/util/List;
// Stack: 3, Locals: 1
public java.util.List compute();

getstatic de.tum.in.ma.extensions.
    CalculationFactory.INSTANCE
ldc "de.tum.in.ma.example.Calculation.
    compute()"
ldc "java.util.List"
invokevirtual de.tum.in.ma.extensions.
    CalculationFactory.get(java.lang.
    String, java.lang.String)
checkcast java.util.List


areturn
```

Table 4.2: The original source code of a method (top left), the corresponding bytecode (bottom left), the bytecode after the mutation (bottom right) and the reverse engineered source code of the mutant (top right).

program to be not terminating.
Consider for example an implementation of an iterator with a method which returns whether further elements are available after the current element. If the method is mutated in such a way that it always returns true, a testcase using the iterator will never terminate.
The selected timeout policy is as follows: The timeout is composed of a constant and an additional variable time which is multiplied with the number of testcases. An absolute upper limit may restrict the calculated time.

4.6 Asymptotic Runtime Complexity

This section discusses the asymptotic runtime complexity of the written prototype. The realistic runtime is presented in the case study for the analyzed study objects (see chapter 6.4.4).

The overall runtime complexity of the workflow is composed of the complexity of the four execution steps and is influenced by the following factors:

- M: number of methods
- R: number of mutants generated per method
- T: number of testcases

Step 1, the instrumentation, goes through the whole project under analysis and instruments all methods and all testcases. It does not take into account whether the methods are under test. The instrumentation of a method or testcase is carried out by inserting a single statement into its code to invoke a logger. It can be performed in constant time. Thus, the resulting worst case runtime complexity is $\mathcal{O}(M + T)$. It is equal to the best-case runtime complexity.

Step 2, which reveals the associations between the methods and testcases, executes each testcase exactly once. A testcase may invoke all methods (directly or via another method). Even though it may invoke a single method more than once, the number of invocations per method should be seen as constant. Hence, one testcase runs in linear time with respect to the number of methods.

The execution of all testcases results in a complexity of $\mathcal{O}(M * T)$. It is the whole complexity of step 2, because the post processing of the collected information, i.e. the transformation of the associations, is negligible.

Step 3 generates mutants of given methods. The effort for the generation of a single mutant is considered to be constant. The number of mutants per method is denominated by R and may vary: Exactly one mutant is generated for a void method, two mutants are generated if the return type is primitive or string, and no or one mutant is generated if an object needs to be returned (no mutant applies if the factory cannot provide a suitable instance). Consequently, R never exceeds two in this approach and should thereby be taken as constant.
The number of considered methods for mutant generation may be below M (number of methods in total) for two reasons: First, some methods are not under test, i.e. they are not covered by at least one testcase. Second, some methods are filtered out, because they are too trivial (e.g. methods returning just a constant value). Nevertheless, this does not asymptotically affect the complexity.
Hence, the complexity of step 3 is $\mathcal{O}(M * R)$ with R being constant.

Step 4 executes the mutants against the testcases. The complexity of running a testcase is specified in step 2 as linear with respect to the number of covered methods. All testcases which cover a mutant need to be processed. In the worst case, all testcases cover all mutants, so that each testcase must be run for each of the $M * R$ mutants. Therefore, the complexity is $\mathcal{O}((M * R) * (M * T))$ equal to $\mathcal{O}(M^2 * R * T)$.

The coordination of the steps of the workflow is considered to run in constant time. Thus, the entire complexity results in $\mathcal{O}(M^2 * R * T)$ respectively $\mathcal{O}(M^2 * T)$ if R is treated as constant. Table 4.3 summarizes the worst case runtime complexities of the workflow:

Step	**Worst case runtime complexity**
1 - Instrumentation	$\mathcal{O}(M + T)$
2 - Information Collection	$\mathcal{O}(M * T)$
3 - Mutation	$\mathcal{O}(M * R)$
4 - Test Execution	$\mathcal{O}(M^2 * R * T)$
Total	$\mathcal{O}(M^2 * R * T)$

Table 4.3: Worst case runtime complexity of the workflow

5 Implementation

This chapter is about the prototypical Java implementation of the approach described in the previous chapter. It covers the architecture, realization details and Java specific peculiarities.

The prototype is implemented in Java and is composed of 122 classes with 535 methods in total. Its source code without tests consists of about 8,100 lines of code (LOC) respectively 6,000 non-comment lines of code (NLOC). 122 testcases in 76 additional test classes ensure the reliability of the executed analysis.
Further extensions and analysis programs (see chapter 5.5) additionally add up to 4,500 LOC.

5.1 Architecture

The program consists of five logical components:
There is one logical component for each of the four steps mentioned in chapter 4. In addition, there is another component, which is in charge of coordinating the whole workflow. It loads the configuration, triggers the workflow steps, provides them with information and manages the interactions between them.

The five components are realized in four Java projects:
The project named *Core* is constructed as a library. It contains utilities and the part of the logic which is needed by more than one project. It is referenced by all other projects and shares the logic.
The project *Main Program* contains the central logic. It is the largest project and comprises the components of workflow coordination, instrumentation and mutation. It is executable and used to start the mutation analysis of a project.
The step *Information Collection* forms an own project and holds the logic of the step of the same name. It is executed by the main program and started as an own process in order to ensure that the modified class files are reloaded.
The step *Test Execution* is also an own project and started in an own process.

Figure 5.1 illustrates the architecture. The rectangles stand for the Java projects; those projects which have a blue border are executable. The packages represent the logical components.

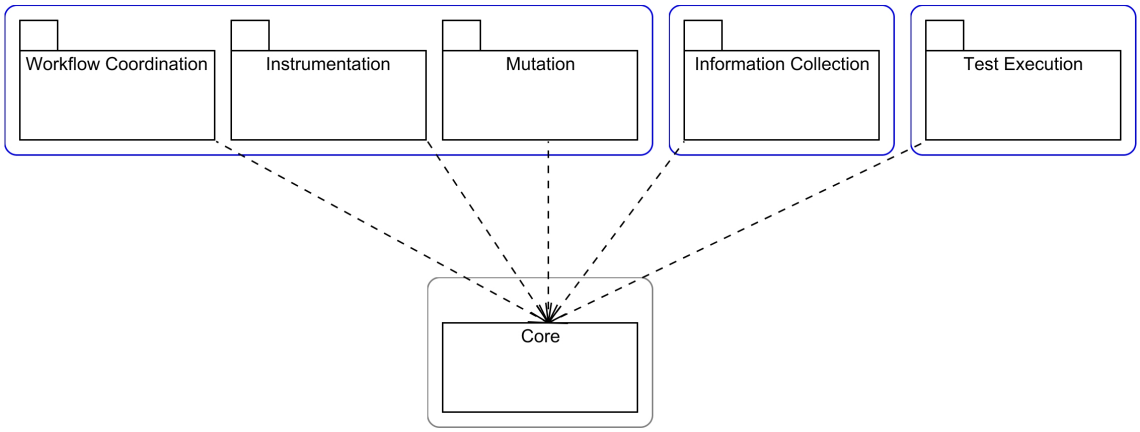


Figure 5.1: Distribution of the logic in projects

5.2 Used Libraries

The following established libraries were used in the prototypical implementation: ASM, log4J, JUnit / TestNG

ASM

ASM[1] is a library which allows analyzing, generating and transforming compiled Java classes. ASM 4.0 was designed to be as small and fast as possible. It is one of the most recent and efficient frameworks for this purpose and supports Java in its latest version (Java 7).
The library provides two different approaches: The core API allows an event based analysis and modification, in which each class is represented as a sequence of events. An event stands for a class element, like for example a field, a method or an instruction, and is realized as a method which can be overwritten. The other approach is object based. A class is represented as a tree of objects. The objects describe the structure and behavior of the classes and are connected through references. This approach is provided by the tree API and built on top of the core API. [38]

ASM is used in the prototype to carry out the bytecode modifications during the instrumentation and mutation and to gather additional information for the empirical evaluation of the results (e.g. to obtain the length of the methods).

Log4j

Apache Log4j[2] is a library providing a logging service. Log4j 2.0 was released in 2012 and provides significant improvements compared to its previous version.

The prototype uses Log4j to track the execution of the analyses. The logs are written both to the console and to a file and contain information about the state during the execution. Thus, they allow following the progress, checking the correctness of an execution and analyzing potential occurring errors.
Log4j addresses the logging aspect, but is not relevant for the computation of the results. Note that the logger mentioned in the steps of instrumentation and information collection is of a separate concern. It is realized as Java class and does not make use of Log4j.

JUnit

JUnit[3] is a popular testing framework for Java projects. It is integrated in the Eclipse IDE and allows executing testcases automatically and repeatedly. Version 4 is shipped with the hamcrest[4] library which permits the use of declarative match rules (constraints or predicates).
JUnit tests can either succeed or fail. The latter case is either caused by unsatisfied assertions (denoted as failure) or by thrown exceptions in the invoked methods (denoted as error).

JUnit is employed in the prototype as test runner for projects which use it as testing framework. Furthermore, numerous unit and system tests were developed with JUnit to ensure the reliability of the prototype.

TestNG

TestNG[5] is another testing framework for automated tests. It supports the concepts of JUnit and introduces further functionalities in the range of parallelization, parameterization and test grouping. It used annotations prior to JUnit.

TestNG is deployed as test runner for projects which rely on it.

1 http://asm.ow2.org/
2 http://logging.apache.org/log4j/
3 http://junit.org/
4 http://code.google.com/p/hamcrest/wiki/Tutorial
5 http://testng.org/doc/

5.3 Aspects

This section comprises some implementation aspects.

5.3.1 Execution in Processes

The program first loads the configuration. Then, the program restarts itself in a new JVM process with the original classpath extended by the one specified in the configuration. This main process performs the bytecode manipulations, while the test executions are executed in own processes.
The processes for running the tests ensure that the modified class files (instrumented or mutated) are used instead of the original ones. This is realized by putting the artifacts which hold the modified files in an earlier position on the classpath than the original ones. Furthermore, processes render the enforcement of timeouts possible. While stopping Java threads goes along with many limitations, processes can easily be terminated. Besides, processes offer the possibility of parallelization.
All processes run in a specified working folder. This allows the provision of necessary test data in form of files.

5.3.2 Use of Jar Files

The program operates on jar files. It takes one or more jar files as input for classes with methods to be mutated and for classes with tests. The instrumented and mutated classes are also packed again in jar files.

5.3.3 Configuration

The configuration allows specifying how the mutation testing analysis shall be performed for a project. The program will either use the command line arguments (if specified when starting it), load a configuration file or request the parameters via console in- and output.
The configuration comprises 18 parameters. Some of the parameters are compulsory (such as the working folder and the jar files with the methods to be mutated), some are optional (such as the test classes to skip and the number of concurrent threads to use). Default values are used for not specified optional parameters.

A sample configuration looks like this:

```
workingFolder = E:/TestExecution/project

jarsWithMethodsToMutate = ./program.jar; ./further_source.jar
jarsWithTestsToRun = ./program.jar; ./further_tests.jar
classpath = ./dependency.jar

returnValueGeneratorNames = DEFAULT
testClassesToSkip = ^org\.tests\.Broken(\w)*$
skipSettersAndGetters = true

testRunner = de.tum.in.ma.logic.runner.junit.JUnitTestRunner
testingTimeoutConstant = 15
testingTimeoutPerTestcase = 10
testingTimeoutAbsoluteMax = 300

numberOfThreads = 4
executeCollectInformation = true
executeMutateAndTest = true
resultPresentation = DB
```

5.3.4 Detection of Tests

The detection of test classes and testcases depends on the used testing framework. The implementation of the detection logic follows the strategy pattern. Plugins can add the support for custom test runners and further testing frameworks.

JUnit 3 Test Classes

JUnit 3 test classes need to inherit `junit.framework.Testcase` or a subclass of it. Testcases are methods in test classes whose name starts with "test".

JUnit 4 Test Classes

JUnit 4 test classes are not required to inherit another class. Methods are considered to be testcases if they are annotated with `@org.junit.Test`. A class will be taken as test class if it contains at least one testcase or inherits another test class.

JUnit Test Suites

Besides the bundling of tests, JUnit 3 test suites additionally allow the dynamic creation of testcases at runtime. A JUnit 3 test suite is a class which implements `junit.framework.Test` (or inherits `junit.framework.TestSuite`) and contains a static method named "suite". The method returns the testcases.
JUnit 4 marks suite classes through annotations. Though, dynamic test behavior is achieved through custom test runners.
In most cases, suites just bundle tests. They are considered only if they return dynamically created testcases.

TestNG Test Classes

TestNG test classes can be found by seeking classes with methods annotated with `org.testng.annotations.Test`. Classes inheriting a detected TestNG test class are considered as test classes, too.

5.3.5 Instrumentation of Testcases

The purpose of the instrumentation of testcases is to enable the logger directly before the execution of the testcase and to disable it immediately afterwards. This has been realized by introducing a try-finally block. It ensures that the latter statement to switch the logger off is always executed before leaving the method (for every possible execution path).

5.3.6 Expandability

The plugin architecture allows extending the application area of the prototype. It facilitates the use of other testing frameworks than JUnit or TestNG by allowing the provision of adapters in plugins. Plugins also allow testing projects which use custom test runners. It is even possible to provide and use another analysis workflow.

5.4 Special Cases

This sections describes some special cases which needed to be considered in the implementation.

5.4.1 Java Synthetic Methods

The JVM Specification states: "Any constructs introduced by a Java compiler that do not have a corresponding construct in the source code must be marked as synthetic, except for default constructors, the class initialization method, and the values and valueOf methods of the Enum class." [39]
Compilers introduce *synthetic methods,* because they are needed for the realization of generics, inner classes, etc. In the following, the characteristics of synthetic methods and their consequence on the mutation testing process are exemplarily described for the generics realization.

Generics in Java are available only at compile time and used by the compiler to ensure type correctness. They are not present in the bytecode. However, the use of generics requires the compiler to introduce helper methods (called *synthetic bridges*) in certain cases.

Consider the following code example. The interface `Data` uses `X` as placeholder for a type. `IntegerData` implements the interface and specifies `Integer` to be the concrete type.

```
public interface Data<X> {
    public void setValue(X value);

    public X getValue();
}


public class IntegerData implements Data<Integer> {
    @Override
    public void setValue(Integer value) {
      // ...
    }

    @Override
    public Integer getValue() {
      return magic();
    }
}
```

Listing 5.1: Synthetic bridge example: interface `Data` and class `IntegerData`

When translating the class to bytecode, the compiler will add two bridges to the class `IntegerData`. They satisfy the interface for the most generic type `Object`. The bridge for the setter method casts the parameter value to the specified type and invokes the original method. The bridge for the getter method invokes the original method and returns its result. Presented as source code, the bridge methods in `IntegerData` look as follows:

```
public void setValue(Object value) {
  setValue((Integer) value);
}

public Object getValue() {
  return getValue();
}
```

Listing 5.2: Synthetic bridge example: bridge methods of the bytecode presented as source code

The consequence of the second bridge is that the bytecode can contain methods which have the same signature (since the return type is not part of it).

When performing mutation testing, synthetic methods need to be considered for the following reasons: First, the mutation of both methods creates an unnecessary overhead. All testcases for methods with a bridge are run twice. Second, it is not any longer possible to use the signature of a method to uniquely identify it. Third, the mutation of a bridge might lead to unexpected results, because in some cases it might not be clear, why a method was chosen for mutation: Consider for example a factory (used to generate instances for returning non-primitive values) which supports the type `Object` but not the (wrapper) type `Integer`. One might be surprised, why the program will mutate the getter method of the code example above, even though its actual return type is not supported by the factory.
Because of these reasons, the prototype identifies synthetic methods through a flag in the bytecode and skips them.

5.4.2 Inherited Testcases

A few projects define some testcases in abstract classes. These testcases often make use of other declared abstract methods to retrieve information such as settings, input and output data for the test execution from them. A couple of classes inherit the abstract test class and specify these parameters by implementing the abstract methods. For this reason, the prototype renders it possible to include testcases inherited from (abstract) classes in the analysis.
Listing 5.3 exemplifies an abstract test class.

```
public abstract class AbstractComputationTest {
    @Test
    public void testComputation() {
      Configuration configuration = getConfiguration();
      Object input = getInputData();

      Object result = Computation.compute(configuration, input);

      assertEquals(getExpectedResult(), result);
    }

    public abstract Configuration getConfiguration();

    public abstract Object getInputData();

    public abstract Object getExpectedResult();
}
```

Listing 5.3: Abstract test class

5.5 Further developed Extensions and Analysis Tools

Additional plugins were implemented for the prototype in order to support other testing frameworks or custom test runners. One plugin for example supports the TestNG framework. It facilitates the detection of TestNG classes, the collection of the contained testcases and their execution.

Furthermore, several analysis tools were implemented for the collection of statistical data.
They ...

- compute the number of statements per method.
- compute the number of statements per testcase and count their assertions.
- perform a static analysis to determine the frequency of the declared return types of methods.
- perform a dynamic analysis to determine the frequency of the actual type(s) of the returned value(s) / instance(s) of a method at runtime.
- parse XML data about the code coverage of methods and projects computed by EclEmma[6].
- simplify the creation of configurations for the mutation prototype and try to automatically assemble the needed classpath for a certain project.
- estimate the needed computation time for the mutation analysis of a certain project.

6 http://www.eclemma.org/

6 Case Study

This chapter contains a case study, which uses the prototypical implementation to collect data from real world software projects. The obtained data is evaluated and the results are presented, analyzed and discussed.

6.1 Research Questions

The case study investigates the following research questions presented in three groups.

In sum, we want to figure out, how reliable the code coverage is, whether metrics exist to reveal inadequately tested methods and weak testcases, and how well the prototype performs compared to other mutation testing tools.

6.1.1 Method Mutation Score

Questions bundled in this group are related to the method mutation score.

RQ 1.1: What is the ratio of inadequately tested methods?

Research question 1.1 reveals the ratio of inadequately and not inadequately tested methods in the study objects. The answer to this question contributes to the assessment of how reliable the code coverage as criterion in general is.

RQ 1.2: Does the method mutation score depend on the type of tests?

This question investigates differences in the ratio of unit and system tests. Methods in study objects with system tests could possibly be less accurately tested, because system tests execute lots of methods in one run. Thus, we expect the ratio of inadequately tested methods in study objects with system tests to be higher than in libraries with unit tests.

RQ 1.3: What category do the inadequately tested methods belong to?

This question analyzes the intended purpose of the methods which are inadequately tested and groups them by this criterion. It sheds light on the type of these methods and thus allows judging their impact and importance.

6.1.2 Factors influencing the Method Mutation Score

The questions in this group investigate the characteristics of methods and testcases as well as the relationships between them. We try to find factors, which can predict inadequately tested methods or weak testcases.

Related to Methods

Research questions 2.1 - 2.3 are related to methods.

RQ 2.1: Does the mutation testing result of a method depend on its return type?

This question examines the correlation between the mutation testing result of methods and their return type. Are methods which return a value less likely to be tested inadequately? What about void methods?

RQ 2.2: Does the mutation testing result of a method depend on its length?

The mutation removes the whole logic of a method and this is presumably more severe for longer

methods. Thus, mutants of longer methods might be easier to be detected as faulty, while mutants of shorter methods might be more likely to stay undetected. Can we find any evidence for or against this assumption?

RQ 2.3: Does the mutation testing result of a method depend on its internal coverage?

The question assesses whether a significant correlation exists between the coverage within a method and the likelihood that it is inadequately tested.

Related to Testcases

Research questions 2.4 - 2.6 are related to testcases.

RQ 2.4: Does the score of a testcase depend on its length?

This question intends to bring to light whether the length of testcases influences their score. Do longer testcases reveal proportionately more of the mutated methods they execute?

RQ 2.5: Does the score of a testcase depend on its number of assertions?

The question examines if the score of testcases is influenced by their number of assertions. Testcases without any or with very few assertions (or other types of checks) might increase the code coverage, but will presumably detect hardly any mutants.

RQ 2.6: Does the score of a testcase depend on its assertion density?

The question investigates if a testcase with more assertions relatively to its length is more successful in detecting inadequately tested methods. If the assumption can be affirmed, test developers could target their improvements especially at testcases with a low assertion density.

Related to Relationships between Methods and Testcases

Research questions 2.7 and 2.8 are related to the interactions between methods and testcases.

RQ 2.7: Does the mutation testing result of a method depend on the number of tests passed through?

Are methods which are covered by many testcases less likely to be inadequately tested? Or does the contrary apply, because such methods are possibly just often executed by the testcases along the way without any result assertions?

RQ 2.8: Does the score of a testcase depend on the number of methods it invokes?

Are testcases which invoke few methods better at detecting mutants? We examine if the score of a testcase goes along with the number of methods it covers.

6.1.3 Comparison with other Mutation Testing Tools

The questions in this group compare the developed prototype with other mutation testing tools.

RQ 3.1: How does the prototype perform compared to other mutation testing tools?

Within this question, we examine the applicability of selected mutation testing tools on different study objects and compare the computed results with focus on the method mutation score and the relevance of the revealed methods. We also have a look at the needed runtime.

RQ 3.2: Can the computed results be reasonably combined with the ones of other tools?

The question analyzes, whether a hybrid solution, which merges the results of two or more tools, would be sensible and lead to better results.

6.2 Study Objects

This section describes the study objects.

We chose different open source projects which satisfy these criteria:

- use of Java as programming language
- public availability of tests
- use of a common testing framework such as JUnit or TestNG

The chosen projects can be classified into libraries, which contain unit tests, and into systems, which contain system tests (and partially unit tests as well). They are of different sizes: The smallest project contains about 7,000 lines of code (LOC), the largest one measures about half a million lines.[1]

6.2.1 Projects with Unit Tests

The following libraries contain code tested by unit tests.

Name	LOC	Website
Apache Commons Collections	109,415	`http://commons.apache.org/proper/commons-collections/`
Apache Commons Lang	100,422	`http://commons.apache.org/proper/commons-lang/`
Apache Commons Math	275,570	`http://commons.apache.org/proper/commons-math/`
Apache Commons Net	53,601	`http://commons.apache.org/proper/commons-net/`
CCSM Commons	43,419	`http://conqat.cqse.eu/`
ConQAT Engine Core	27,481	`http://conqat.cqse.eu/`

Table 6.1: Projects with unit tests

Apache Commons Collections

Commons Collections is a library by the Apache Software Foundation. It extends the data structures provided by the Java JDK and adds further interfaces, implementations and utilities.
Its 109,415 lines of code are tested by more than 4,400 testcases. Hence, it is the study object with the most testcases.

Apache Commons Lang

The Apache library *Commons Lang* extends the Java standard libraries by providing more methods to manipulate core classes. It contains helper methods for string manipulations, basic numerical operations, concurrency, etc.
We used version 3.1. It has 387 classes with 4,651 methods and 2,000 JUnit testcases.

Apache Commons Math

Apache Commons Math is a project which contains functionalities to address common mathematical and statistical problems.
Version 3.1 consists of 1,469 classes and contains 275,570 lines of code. It uses JUnit as testing framework and is shipped with 3,463 testcases.

Apache Commons Net

Apache Commons Net is a web-related library. It contains the implementations of several internet protocols such as FTP, SMTP, IMAP, etc.
The project's size is 53,600 lines of code. We used 133 unit tests. The project contains more, but not all of them could be executed with success on the original code base.

1 The lines of code reflect the size of a project including its tests. The project may depend on other libraries whose size is not included.

CCSM Commons

CCSM Commons is a library of the ConQAT project, a software quality analysis engine developed by the Technische Universität München. CCSM Commons contains utility functions related to the string type, the file system, reflection, etc. It was included in some Unix distributions.
We used the release 2012.9 in which 468 testcases check about 43,000 lines of code.

ConQAT Engine Core

ConQAT Engine Core is another library of the ConQAT project and is built on top of CCSM Commons. It contains drivers and processors needed for the code analysis process.
Release 2012.9 has about 27,000 lines of code. 107 JUnit tests ensure the quality.

6.2.2 Projects with System Tests

The following projects are systems which contain system tests. They usually have less tests, but the execution of a single one covers a larger portion of the system.

Name	LOC	Website
ConQAT dotnet	8,239	`http://conqat.cqse.eu/`
DaisyDiff	11,288	`http://code.google.com/p/daisydiff/`
Histone	24,412	`http://weblab.megafon.ru/histone/en/`
LittleProxy	7,300	`http://github.com/adamfisk/LittleProxy/`
Mutation Testing Prototype	11,231	-
Predictor	7,733	`http://code.google.com/p/predictor/`
Struts 2	148,486	`http://struts.apache.org/`
Symja	443,092	`http://code.google.com/p/symja/`
Tspmccabe	44,999	`http://code.google.com/p/tspmccabe/`

Table 6.2: Projects with system tests

ConQAT dotnet

ConQAT dotnet is also a part of the ConQAT analysis software. It comprises processors to analyze Microsoft .NET projects. Actually, it is not a pure system test project. However, the tests can be considered as close to system tests since they trigger the execution of external code written in .NET and check the final result.
22 tests cover 896 Java methods of the program itself and of underlying ConQAT parts (CCSM Commons, Engine Core, and others).

DaisyDiff

The project *DaisyDiff* was developed during the "Google Summer of Code" program in 2007. Its purpose is to compare two HTML files, calculate the difference and present the result. The result is a new HTML file, in which added, removed or differently formatted text pieces (characters, words or paragraphs) of one input file compared to another are highlighted.
We considered only the testcase `org.outerj.daisy.diff.html.FileBasedTest.test()` to be a system test. However, the testcase is parameterized and invoked with more than 200 different input data sets: It takes two files to be compared and the expected result as input and checks the equality of the computed and expected result.

Histone

Histone is a powerful template engine for generating code in HTML and other text formats. Besides the Java implementation, versions for PHP and Javascript exist and allow a cross-platform use. It is fast, because it compiles the templates once and avoids the parsing procedure at every use.

The 93 testcases in the package `ru.histone.acceptance` were considered as system tests. They cover about 700 methods. Further unit tests extend the code coverage.

LittleProxy

LittleProxy is a high performance HTTP proxy written in Java. The project is built on top of an event-based network library called Netty. The developers consider it to be performing well and to be easily integrable into other projects.
With 7,300 lines of code it is the smallest study object.

Mutation Testing Prototype

The *mutation testing prototype* developed as part of this thesis is also a study object. To facilitate the analysis of this program with itself, some adjustments are needed. The adjustments are described in section 6.4 of this chapter.
The used version was mature and consisted of 11,000 lines of code. 96 unit and 5 system tests were used.[2]

Predictor

The *Predictor* project is an open source framework for genetic algorithms (GA). The developers describe genetic programming as being "based on the idea of expressing algorithms [...] as a set of operations which can mutate, breed, and compete with other algorithms".
The project looks for algorithmic patterns within a stream of data. It is not restricted on a particular domain and will eventually support distributed algorithms. The developers aimed to develop a solution, which is not overly academic and can be used in enterprises.
The project contains both unit and system tests. All classes inheriting `com.intermancer.predictor.system.SystemTest` are considered as system tests.
Predictor is the only project which uses TestNG as testing framework. All other study objects use JUnit.

Struts 2

Apache Struts is a well known open source web framework making it easy to write Java web applications. It permits the separation of concerns by providing the model-view-controller (MVC) approach.
We used version 2.3.14.3 which contains sample projects demonstrating how to use the framework. The sample project called *portlet* contains web testcases, which send requests and check the response. They invoke lots of methods in the Struts framework and can thereby be seen as system tests.

Symja

Symja is a Java computer algebra system with the ability to parse, evaluate and plot mathematical expressions. It supports polynomials, differentiation, pattern matching, linear algebra, complex numbers etc. The project was forked from MathEclipse[3], consists of about 450,000 lines of code and is thereby the largest project.
The test classes in the package `org.matheclipse.core.system` were taken as system tests. Their 448 system tests check the correctness and cover 3,025 methods.

Tspmccabe

Tspmccabe analyzes classes or entire programs written in Java. It builds an abstract syntax tree and calculates, among other metrics, the McCabe complexity[4] on method and class level.

2 The key figures differ from the ones mentioned in the implementation chapter, because they regard different versions. The lines of code include the test code, too.
3 `http://www.findbestopensource.com/product/symja`
4 `http://www.literateprogramming.com/mccabe.pdf`, 01.09.2013

The program consists of 175 classes with 739 methods, which result in 50,000 lines of code. The class `tspmccabe.tests.SystemTests` targets at comparing the expected with the computed result for the given test data.

6.2.3 Gained Experiences

The selection and preparation of study objects was straightforward for libraries, but not so easy for system projects. We had a look at many systems, however, a high number of them could not be used for several reasons:

- some systems could not get built with reasonable effort
- some were depending on missing libraries which were no longer available
- some were depending on other remote systems
- some were using libraries in a version not compatible to the ones of the prototype
- some had system tests out of which the majority was not working
- some had system tests which needed a very high effort in terms of configuration
- some had testcases (and the source code too) of an unacceptable quality (e.g. plenty of code commented out)

Even though owners of some projects were contacted, it was mostly not possible to fix the issues. A couple of people responded that the existing system tests were not working, because they were no longer maintained and used in the build process.

6.3 Case Study Design

We plan to use the prototype to execute the mutation analysis for different study objects. Then, we want to analyze the obtained data to answer the research questions. The design of the research questions is as follows:

6.3.1 Method Mutation Score

This section describes the design of research questions related to the method mutation score.

RQ 1.1: What is the ratio of inadequately tested methods?

The terms *inadequately tested method* and *method score* are defined in chapter 2.1.2. The number of *processed methods* is defined in this context as the number of methods for which at least one mutant was generated and executed (without timeout).
We calculate the ratio as the number of inadequately tested methods divided through the number of all tested methods. Note that easily detectable equivalent mutants have already been excluded. If, for example, 50 methods are inadequately tested out of 1,000 processed methods under test, the ratio of inadequately tested methods is 5%.

RQ 1.2: Does the method mutation score depend on the type of tests?

In order to answer this question, we examine the method mutation scores of unit and system tests separately and compare them.

RQ 1.3: What category do the inadequately tested methods belong to?

We classify inadequately tested methods by their purpose and group them by their severity:

- **Irrelevant:** Methods which are not deterministic (e.g. the generation of a seed value) or methods which are covered but not intended to be tested (test helper methods) belong to this group.
- **Low:** Methods which usually do not significantly influence the program execution are in this group. Among them are methods to close streams (mostly the rest of the program will still work if a stream accidentally remains open), optimizations (e.g. condensing a tree), methods for validations (e.g. to check the validity of a parameter), for logging outputs, for

performance dumps, etc.
Furthermore, the `hashCode()`[5] method belongs to this group. Unless the generated hash code is explicitly subject of the test, tests covering this method will not be able to detect the mutation, because the method's behavior is still in correspondence with its specification (since always the same constant value is returned).

- **Medium:** Methods which might be relevant for the program execution belong to this group. One example are `toString()` methods. Methods performing preparatory and follow up operations for workflows are also counted in.
- **High:** Methods which are considered to be highly relevant for the execution belong to this group. They are tested less accurately than they should be. Examples are: `calculateAST()`, `writeToFile()`, `putObject()`, ...

The classification is arranged on the name of the method. It is partially automated by using regular expressions for common method names.
In case of doubt, we choose the more severe group.

6.3.2 Factors influencing the Method Mutation Score

This section describes the design of research questions which investigate possible influence factors on the method mutation score.

RQ 2.1: Does the mutation testing result of a method depend on its return type?

The term *mutation testing result of a method* is defined in section 2.1.2.
We examine whether a correlation exists between the groups of return types (void, primitive and string, object) and the mutation testing result.

RQ 2.2: Does the mutation testing result of a method depend on its length?

We take the number of bytecode instructions as the length of a method. We check whether a correlation between the method length and the mutation testing result can be detected.

RQ 2.3: Does the mutation testing result of a method depend on its internal coverage?

We analyze the detailed coverage of a project with the Eclipse plugin EclEmma. Thereby, we get the individual instruction and branch coverage for each method. We set the coverage in relation to the mutation testing result of the method.

RQ 2.4: Does the score of a testcase depend on its length?

The *score of a testcase* is defined in chapter 2.1.2.
The computation of the length of testcases is performed equally to the one of methods: We count the number of bytecode instructions. Within the scope of this question, we examine whether the testcase length and score correlate.

RQ 2.5: Does the score of a testcase depend on its number of assertions?

We count the assertion statements in testcases. The invocations of assertion methods, which are provided by the used testing framework, are considered as well as the invocations of frequently used project specific checker methods.
We set the number of assertions in relation to the test score. We want to find out if the number of assertions allows conclusions to be drawn about the test reliability.

RQ 2.6: Does the score of a testcase depend on its assertion density?

To answer this question we calculate the assertion density of a testcase by dividing the number of assertions by the testcase length. We investigate if the assertion density has a statistically significant influence on the testcase score.

5 The `hashCode()` method in Java returns a hash code value for an object. According to the specification, it must produce the same integer result for objects which are equal in compliance with the `equals(Object)` method. [40]

RQ 2.7: Does the mutation testing result of a method depend on the number of tests passed through?

We count for each method by how many testcases it is executed and determine, if the number has an influence on the mutation testing result of a method.

RQ 2.8: Does the score of a testcase depend on the number of methods it invokes?

We count the number of executed methods per testcase. Subsequently, we examine the significance of the correlation between the number of executed methods and the testcase score.

6.3.3 Comparison with other Mutation Testing Tools

This section describes the design of research questions related to the comparison of the developed prototype with other mutation testing tools.

RQ 3.1: How does the prototype perform compared to other mutation testing tools?

We use other popular mutation testing tools to analyze some of the study objects. Furthermore, we re-analyze these study objects with the prototype with comparable settings (exceptionally with setters and getters, modified timeout settings, no parallelization).
We investigate how many methods are analyzed and compare the aggregated results (absolute number and relative ratio of inadequately tested methods).
Moreover, we map the methods detected by the tools to groups (analogous to research question 1.3). This allows a rough comparison of the relevance of the detected methods between different tools.
Finally, we present the runtime needed by the tools.

RQ 3.2: Can the computed results be reasonably combined with the ones of other tools?

We analyze to which extent scopes and results of the different tools are overlapping. For that, we consider, which methods are covered by the analysis, and examine, which methods are detected by all programs as inadequately tested and which are detected exclusively by one tool. Finally, we evaluate if the benefit of an improved result justifies the additional effort.

6.4 Execution

This section describes the execution of the mutation testing analysis for the study objects in detail in order to permit reproducibility.

6.4.1 Procedure

As first step, we selected study objects, checked out their source code and built them with Ant or Maven according to the instructions. Afterwards, we imported the projects in the Eclipse IDE, had a look at the testcases and run them once.
While the libraries contained only unit tests, some system projects had non system tests, too. We excluded tests which did not correspond to our definition of system test from system projects.
Subsequently, we exported the compiled source and test code as jar files. After that, we figured out the classpath for the execution. Then, we created a configuration file for the mutation testing program. We specified the classpath, the working folder (in which the analysis will be executed and where the test resources are located), paths to the exported jar files, the number of threads to use, patterns to exclude test classes, the used testing framework, etc. in the configuration (see section 6.4.2). Furthermore, we checked whether an adaption or extension was needed to run the tests (see section 6.4.3).
At this point the project was ready for the analysis. We started the mutation prototype, selected the configuration file and let the execution begin. Depending on the size of the study object, the execution took between a couple of minutes and some hours. Section 6.4.4 presents the runtime.
Moreover, we executed other developed analysis tools to collect statistical data about the methods and testcases.

After the completion of the execution, we considered the log file to check whether any unexpected failures had occurred. (If that was the case, we would fix the cause and restart the analysis.) Then, we imported the results, which were in form of SQL statements, into a database. We also stored the configuration and additional execution information (extracted from the log file). Finally, we determined methods which are provable equivalent and excluded them from the further analysis.

6.4.2 Configuration

The following settings were applied to the execution of all projects:

- We used the optional feature of the prototype to skip too trivial methods (see chapter 4.4). The feature to skip empty methods was not supported yet, when the projects were executed. Nevertheless, these methods were filtered out afterwards in the database.
- We specified the timeouts for the test execution in a way that the testcases would even pass if they took more than twice as long as on the original code.
- Projects which are likely to cause another result if executed in parallel were run with only one process. Among them are projects using the network (Apache Commons Net which uses sockets, LittleProxy) and projects which carry out many write operations. We used up to four parallel processes to analyze the other projects.
- All projects were executed in a folder which contained the necessary test data.
- Factories were provided for the projects Commons Collections, Commons Lang and ConQAT Engine Core. These projects were executed once with and once without factory. For all other projects, only methods with void and primitive (including string) return were considered.

6.4.3 Modifications, Adjustments and Extensions

The following modifications, adjustments and extensions were necessary to execute the tests of some projects or to allow a deeper analysis.

Library: Apache Commons Collections

We implemented a factory to support methods with non-primitive return types, too. The factory uses the declared return type as selection criterion for the appropriate instance.

Library: Apache Commons Lang

We implemented a factory to support methods with non-primitive return types. Like the factory for Commons Collections, it uses the declared return type as selection criterion for the appropriate instance.

Library: Apache Commons Math

No special adaption was made.

Library: Apache Commons Net

A couple of testcases of this project were not running on the original code. A majority of them uses network functionalities and thereby might require certain firewall settings. The prototype automatically detected and skipped these testcases. Enough working testcases remained to continue the analysis.

Library: CCSM Commons

No special adaption was made.

Library: ConQAT Engine Core

When analyzing the ConQAT Engine Core project, the test class `DriverOptionsTest` needed to be skipped.

We developed a program for the dynamic analysis of the methods under test to identify the actual class types of the returned instances (rather than the declared ones). Then, we implemented a factory which creates instances for methods with a non-primitive return type. It consists of a switch-case block with 174 cases and uses the method name as selection criterion for the appropriate instance.

System: ConQAT dotnet

Methods in other referenced ConQAT libraries were mutated, too.

System: DaisyDiff

The testcase of the class `FileBasedTest` is parameterized and uses a custom test runner. We developed a plugin to facilitate the use of the custom runner.

System: Histone

The test classes of Histone are annotated with `@RunWith` and use a custom runner to set up and execute the tests. We developed a plugin for the mutation testing prototype to support the tests of this project as well.

It is a system test project, but its additional unit tests were also considered in a separate execution.

System: LittleProxy

We excluded the test classes:

- `org.littleshoot.proxy.HttpsFilterTest` (some of its testcases threw exceptions and were not terminating in the original state)
- `org.littleshoot.proxy.ProxyUtilsTest` (the testcases were considered as unit rather than system tests and LittleProxy is a system test project)

System: Mutation Testing Prototype

In order to be able to analyze the prototype as study object, some modifications were necessary. They were applied to a cloned version of the prototype.

- The classes of the prototype in the role of executing the mutation analysis and in the role of being the study object have the same name. Thus, they were renamed in the cloned version by changing the package names from `de.tum.in.ma.*` to `de2.tum.in.ma.*`.
- The program contains one class, which starts some steps of the analysis in another JVM process. The assembling of the classpath to start the steps required an adaption, so that the mutated class files are listed prior to the original entries.
- The name of the temporary working folder had to be renamed in order to avoid file conflicts during the execution.

We ensured that all tests were still running successfully after the modification of the clone.

It is a system test project, but its additional unit tests were also considered in a separate execution.

System: Predictor

Predictor uses the TestNG framework. We developed a plugin for the mutation testing prototype to support tests using this framework, too.

It is a system test project, but its additional unit tests were also considered in a separate execution.

System: Struts 2

The methods in the referenced and closely tied library *xwork-core* were mutated, too.

System: Symja

No special adaption was made.

System: Tspmccabe

This project contains a test class with a system test which checks the final result of the calculation. A folder with a couple of test data files belongs to the project. However, only one of the files was used by the testcase, even though the others were suitable, too. Therefore, we extended the class so that it considers all test data files and reaches a higher code coverage.

6.4.4 Runtime

Table 6.3 lists the execution duration of the study objects. All executions were run on a machine with an Intel i7-3517U quad core processor with 1.90 GHz and 8 GB RAM.
The *duration* concerns the whole analysis (executed with the specified number of processes), *tests only* refers to the single execution of all testcases (always executed in a single process). The *slow-down factor* is the quotient of the duration of the whole analysis (multiplied with the number of processes) and of the single execution of all testcases.

Note that the runtimes are listed to give a rough impression of the duration. However, the duration cannot be directly compared between different study objects, because it depends on different factors:

- number of testcases
- number of methods under test
- runtime / complexity of the testcases
- average number of testcases per method
- number of used Java processes
- number of mutants causing a timeout

Project	Type	LOC	Testcases	Methods 1)	Java processes	Duration (h:mm:ss)	Tests only (h:mm:ss)	Slow-down factor	Notes
Apache Commons Collections	Library	109,415	4,468	VPS	2	0:59:08	0:00:16	443.5	
Apache Commons Collections	Library	109,415	4,468	instances	2	1:17:21	0:00:16	580.1	
Apache Commons Lang	Library	100,422	2,000	VPS	2	0:51:47	0:00:18	345.2	
Apache Commons Lang	Library	100,422	2,000	instances	2	0:03:17	0:00:18	21.9	
Apache Commons Math	Library	275,570	3,463	VPS	1	12:10:33	0:06:10	141.4	
Apache Commons Net	Library	53,601	133	VPS	1	1:00:39	0:01:02	58.7	2)
CCSM Commons	Library	43,419	468	VPS	4	0:09:37	0:00:03	384.7	
ConQAT Engine Core	Library	27,481	107	VPS	1	0:04:32	0:00:03	14.7	2)
ConQAT Engine Core	Library	27,481	107	instances	1	0:00:44	0:00:06	24.0	2)
ConQAT dotnet	System	8,239	22	VPS	2	1:06:23	0:00:13	612.8	
Daisydiff	System	11,288	1	VPS	4	0:04:46	0:00:02	572.0	3)
Histone	System	24,412	93	VPS	1	0:59:45	0:01:05	55.2	
Histone	System	24,412	99	VPS	1	0:18:08	0:00:03	362.7	4)
LittleProxy	System	7,300	93	VPS	1	0:21:24	0:00:11	116.7	
Mutation Testing Prototype	System	11,231	5	VPS	1	0:18:20	0:00:35	31.4	
Mutation Testing Prototype	System	11,231	110	VPS	1	0:08:27	0:00:02	253.5	4)
Predictor	System	7,733	21	VPS	1	1:31:45	0:00:16	344.1	
Predictor	System	7,733	55	VPS	1	0:04:30	0:00:17	15.9	4)
Struts 2	System	148,486	6	VPS	1	1:30:24	0:00:02	2,712.0	
Symja	System	443,092	448	VPS	2	1:37:40	0:00:10	1,172.0	
Tspmccabe	System	44,999	10	VPS	1	0:43:44	0:00:02	1,312.0	

Table 6.3: Duration of the whole analysis

1) Return type of methods (VPS = void, primitive, string).
2) Executed with a newer optimized version of the prototype, which filters provable equivalent methods already at an early point and thus reduces the testing effort significantly. (The equivalent mutants of the other projects were removed afterwards in the database.)
3) The single testcase of this project is parameterized. It is executed more than 200 times with different input data.
4) Additional execution of a system project which considers only the unit tests.

As already apparent from the asymptotic runtime complexity (discussed in chapter 4.6), the last step of the mutation approach, the *test execution*, demands most of the execution time.
Thus, table 6.4 gives a more detailed insight into the duration of two selected projects. It shows that the performed bytecode manipulations are really fast: The *instrumentation step*, which touches all methods and testcases, needs only few seconds even for larger projects. The *information collection* takes the time needed to run all testcases once. The *mutation* and *test execution* done for each method under test need together more than 99% of the runtime[6]. The mutation itself changes only one single method at a time; it works like the instrumentation on bytecode and hence has a negligible duration. The test execution runs numerous testcases for each performed mutation and is expensive. It is heavily influenced by the chosen timeout settings for some projects. Some time is also spent for the creation of a new Java process for the test execution of each mutant.

	Commons Collections	**Commons Math**
Instrumentation	0:01	0:03
Information Collection	0:16	6:10
Mutation and Test Execution	58:49	723:19
Other	0:02	0:01
Total	59:08	729:33

Table 6.4: Duration (mm:ss) of the different steps of the analysis

6.5 Results and Discussion

This section presents and discusses the results of the research questions.

6.5.1 Method Mutation Score

RQ 1.1: What is the ratio of inadequately tested methods?

This analysis is addressing research question 1.1, the question concerning the ratio of inadequately tested methods. The ratios are calculated only on methods with void, primitive or string return types;[7] furthermore simple setters and getters are skipped (see chapter 4.4).
The resulting ratio varies heavily between different study objects and ranges between 5% and 50% for most projects. The measure certifies the testcases of Apache Commons Lang to be quite effective. The testcases cover 93% of the methods and out of these less than 2% are inadequately tested. The testcases of some other projects do not perform that well. Nearly half of the covered methods of the projects Predictor and Struts 2 are considered as inadequately tested. The result of LittleProxy is even worse, about 71% of the covered methods are inadequately tested. Accordingly, the system tests of LittleProxy test most methods without effect.

Table 6.5 presents the number and ratio of inadequately tested methods for the study objects.

Table 6.6 presents the method mutation score, the code coverage and the product of these two values. The method mutation score is the residual ratio to the inadequately tested methods. The code coverage has been calculated with EclEmma and is based on method level. It expresses which ratio of the methods is covered by tests. The product of method mutation score and coverage corresponds to the ratio of not inadequately tested methods out of *all existing* methods (under the assumption that the covered but not analyzed methods provide a roughly similar result).

Diagrams are presented as part of the next research question.

RQ 1.2: Does the method mutation score depend on the type of tests?

Next, we want to answer, whether the method mutation score of unit and system tests differs. As already apparent from table 6.6, this is effectively the case. The diagrams 6.1 and 6.2 illustrate the method mutation scores separately for libraries and systems.

6 Their duration was not measured separately.
7 Roughly 50% - 60% of the methods belong to this group.

Project	Type	Number of inadequately tested methods	Ratio of inadequately tested methods
Apache Commons Collections	Library	124	9.48%
Apache Commons Lang	Library	22	1.85%
Apache Commons Math	Library	271	10.56%
Apache Commons Net	Library	28	18.42%
CCSM Commons	Library	45	9.25%
ConQAT Engine Core	Library	41	18.92%
ConQAT dotnet	System	154	36.25%
Daisydiff	System	7	6.36%
Histone	System	47	24.83%
LittleProxy	System	35	71.43%
Mutation Testing Prototype	System	7	25.00%
Predictor	System	80	52.74%
Struts 2	System	154	45.85%
Symja	System	234	25.03%
Tspmccabe	System	13	21.31%

Table 6.5: Number and ratio of inadequately tested methods

Project	Method mutation score	Code coverage	Product
Apache Commons Collections	90.52%	81.59%	73.86%
Apache Commons Lang	98.15%	93.03%	91.31%
Apache Commons Math	89.44%	84.78%	75.83%
Apache Commons Net	81.58%	29.03%	23.68%
CCSM Commons	90.75%	56.30%	51.09%
ConQAT Engine Core	81.08%	50.00%	40.54%
ConQAT dotnet	63.75%	48.10%	30.66%
Daisydiff	93.64%	49.82%	46.65%
Histone	75.17%	73.01%	54.88%
LittleProxy	28.57%	45.38%	12.97%
Mutation Testing Prototype	75.00%	73.08%	54.81%
Predictor	47.26%	72.44%	34.24%
Struts 2	54.15%	26.99%	14.62%
Symja	74.97%	21.25%	15.93%
Tspmccabe	78.69%	39.05%	30.73%

Table 6.6: Method mutation score and code coverage of the study objects

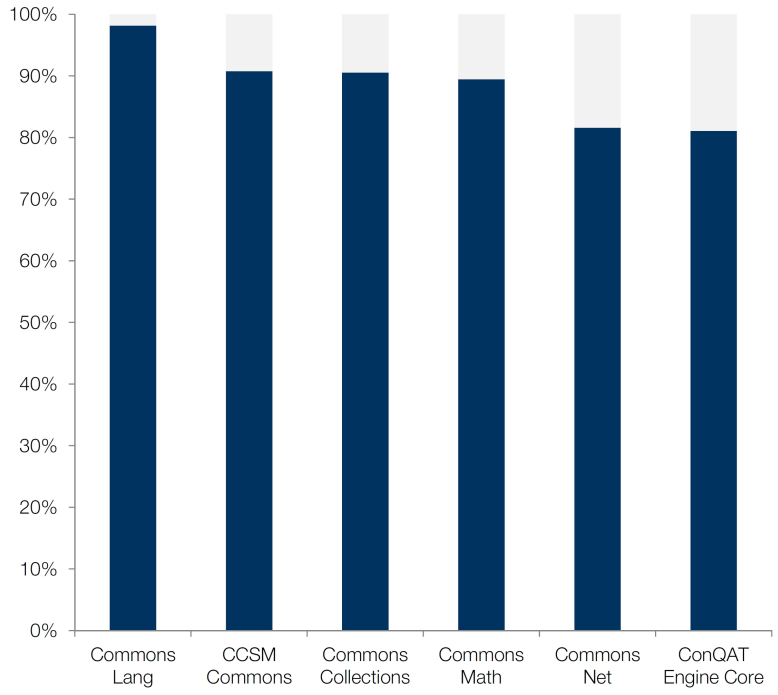


Figure 6.1: Method mutation score of libraries with unit tests

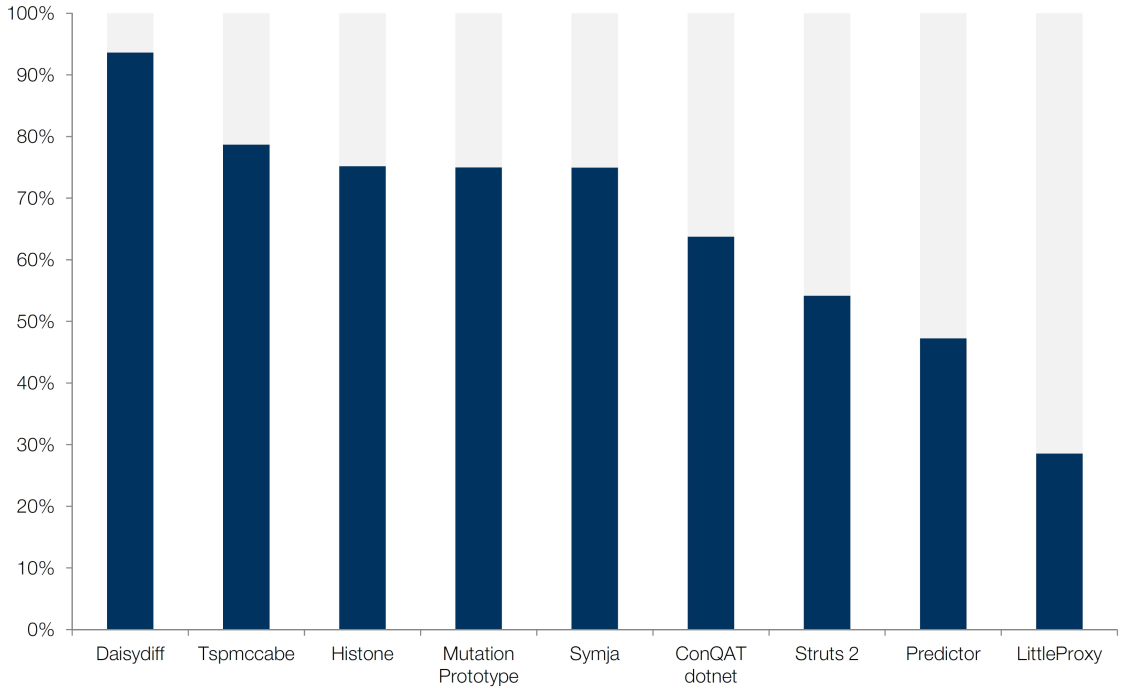


Figure 6.2: Method mutation score of systems with system tests

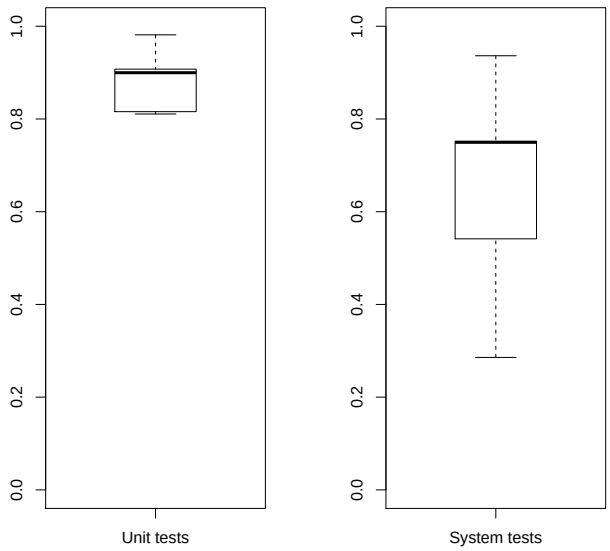


Figure 6.3: Box plot comparing the method mutation score of unit and system tests

The average method mutation score for unit tests is 88.59%. It is higher than the average score of system tests which is 65.69%. The box plot in figure 6.3 depicts the key figures and points the difference in the deviation up. The method mutation scores of system tests deviate heavier among the different projects. Their standard deviation is 19.58% compared to 6.42% in unit tests.

The differences in the method mutation scores between the two test types can be explained by two factors:
First, libraries mostly provide utilities and carry out computations. Systems which mostly perform a calculation (Daisydiff, Tspmccabe, Symja) also tend to achieve a relatively high method mutation score. In contrast, other systems which have a broader scope of application get a lower score. The testcases of the proxy LittleProxy and those of the web framework Struts 2, which test the interfaces of sample websites, are exemplary for systems with a broader scope.
Second, the assumption that system tests invoke more methods and thus check single methods more superficially, applies. The unit tests of the study objects invoke on average 25.66 methods (median: 17.18, standard deviation: 25.34). System tests invoke on average 194.86 methods (median: 162.76, standard deviation: 175.87).

Some systems contain in addition to the system tests further unit tests. They were additionally analyzed for the projects Histone, Predictor and for the mutation testing prototype. Table 6.7 shows that the combination of both test types improves the overall result. The code coverage increases thereby, too.

Project	System tests	Unit tests	Combination
Histone	75.17%	63.93%	79.63%
Mutation Testing Prototype	75.00%	92.25%	91.77%
Predictor	47.26%	80.00%	74.86%

Table 6.7: The combination of unit and system tests improves the overall method mutation score

RQ 1.3: What category do the inadequately tested methods belong to?

We address research question 1.3 by classifying the inadequately tested methods by their assumed purpose and then grouping them by their severity. We make the following observations:
Less than 1% of the revealed methods are classified on average as irrelevant. They are either not

deterministic and hence mostly not sensibly testable or belong to the test infrastructure[8].
About 10% - 30% (depending on the project) of the inadequately tested methods are on average considered as uncritical (low severity). We expect them not to have a significant influence on the program execution.
The rest of the methods are of medium or high severity. These methods are supposed to be relevant for the study object. They do, but should not, appear in the list of inadequately tested methods. The coverage marks them as tested and thus pretends a false sense of correctness.
Figure 6.4 illustrates the results for some study objects. Figure 6.5 shows a more detailed categorization exemplarily for the project Apache Commons Math. The numbers indicate the methods per category.

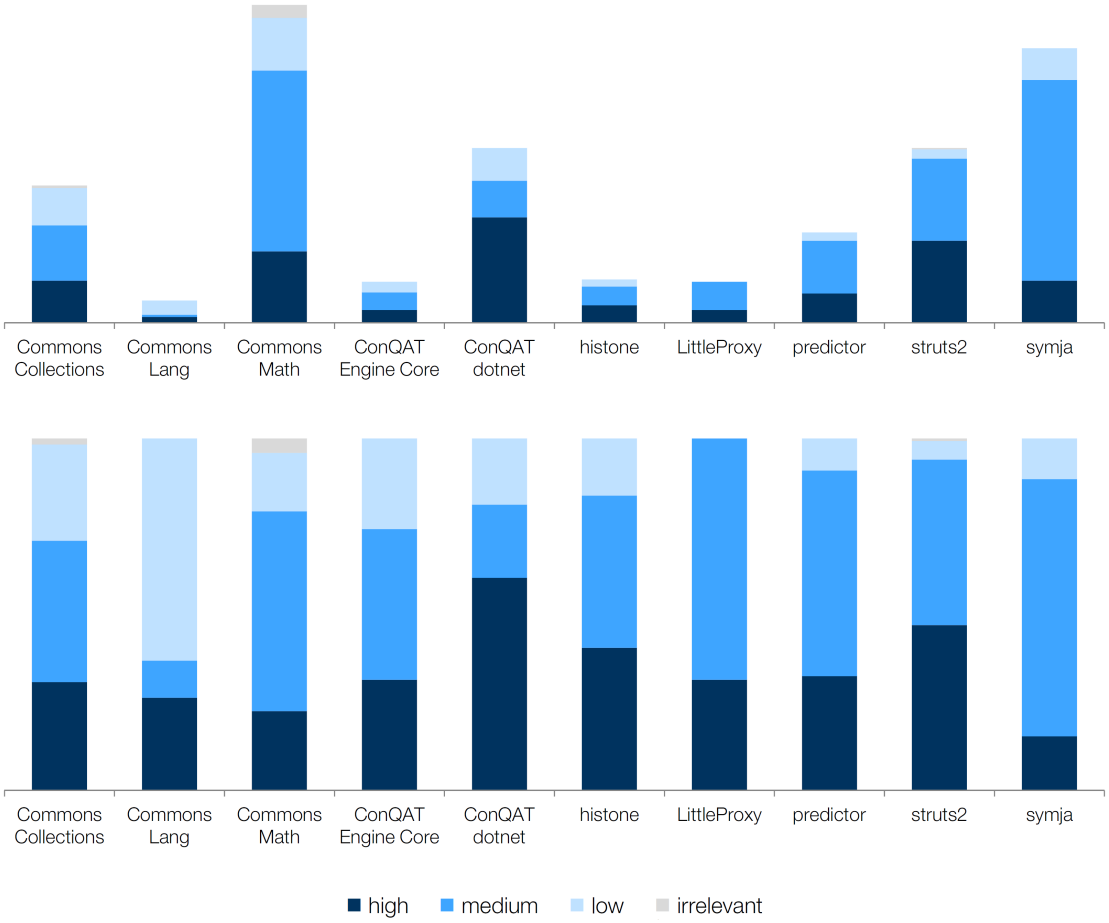


Figure 6.4: Classification of inadequately tested methods by their severity (top: absolute number, bottom: ratio)

6.5.2 Factors influencing the Method Mutation Score

RQ 2.1: Does the mutation testing result of a method depend on its return type?

This section answers, whether the method mutation score is influenced by the return type of the methods.
Table 6.8 supplements the results presented in research question 1.1. It shows the analysis results of three selected projects in which only methods returning objects were considered. The underlying mutation analysis was enabled by the use of three project specific factories (see chapter 6.4).
Figure 6.6 shows the method mutation score dissected by the return type groups. The letters denote these groups: “V” stands for void, “P” for primitive and string, “OBJ” for objects. The

8 Some few classes providing test utilities are not always cognizable as such. Those which are cognizable are not instrumented and thus not included in the analysis.

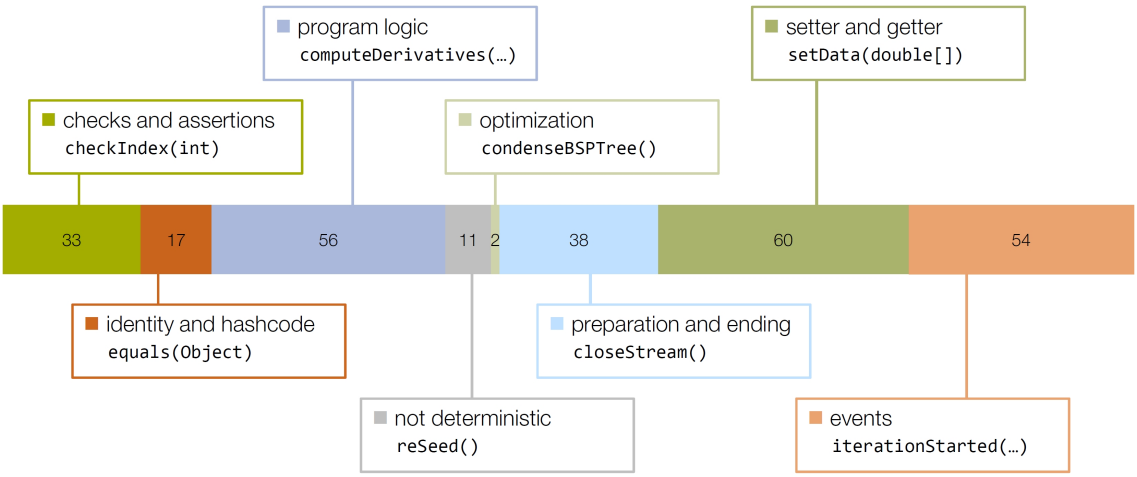


Figure 6.5: Categories of inadequately tested methods (Apache Commons Math)

Project	Type	Ratio inadeq.	Method mutation score
Apache Commons Collections	Library	1.41%	98.59%
Apache Commons Lang	Library	0.51%	99.49%
ConQAT Engine Core	Library	18.68%	81.32%

Table 6.8: Method mutation score for selected study objects considering only methods which return objects

colored bars indicate the deviation from the overall method mutation score of the project. The red bar expresses, how much worse the method mutation score of the considered methods compared to the average is, the green bar respectively shows the positive deviation.

Except for ConQAT Engine Core, the method mutation score of methods with void return is in all projects (including the not illustrated ones) significantly lower than of methods which return a value. Hence, void methods seem to be more likely inadequately tested. In contrast, methods which return a primitive value are on average less likely inadequately tested.
This applies to methods which return objects too and stands especially out for the Apache Commons Collections project (methods returning objects are 5.7% above the overall score). However, we presume that this might be caused by the factory. The factories for the two Apache projects select an instance by considering the return type. Thus, if the invoker of a method expects a more specific type (i.e. a more specialized class) than the declared return type, a possible class cast will result in a cast exception causing the testcase to fail.

RQ 2.2: Does the mutation testing result of a method depend on its length?

First, we had a look at the distribution of the method length of different projects and identified, that the majority of the methods consists of only few instructions.
Then, we applied hypothesis testing to analyze the data: The one-tailed null hypothesis H_0 is that inadequately tested methods are as long as or longer than the other methods. Consequently, the alternative hypothesis H_1 is that inadequately tested methods are shorter. We used the Wilcoxon signed-rank test, because the data is not normally distributed.
H_0 can be rejected for 6 study objects under the significance level of 5% (Apache Commons Math: $p < 0.001$; Struts 2: $p < 0.001$; Symja: $p < 0.001$; CCSM Commons: $p = 0.002$; ConQAT Engine Core: $p = 0.019$; Predictor: $p = 0.025$), so that the alternative hypothesis H_1 is appropriate for them. H_0 cannot be rejected for the three projects Apache Commons Collections, ConQAT dotnet and Histone, because their p-value is greater than 5%. The remaining 6 projects had only a low absolute number of inadequately tested methods, therefore we excluded them from the analysis in order to keep it statistically sound.[9]

Figure 6.7 demonstrates the result of the project Symja by illustrating the histogram of the method length. The red bar indicates the distribution of the inadequately tested methods, the green bar the distribution of the not inadequately tested ones. The ratio of inadequately tested methods is,

9 We excluded projects with less than 40 inadequately tested methods.

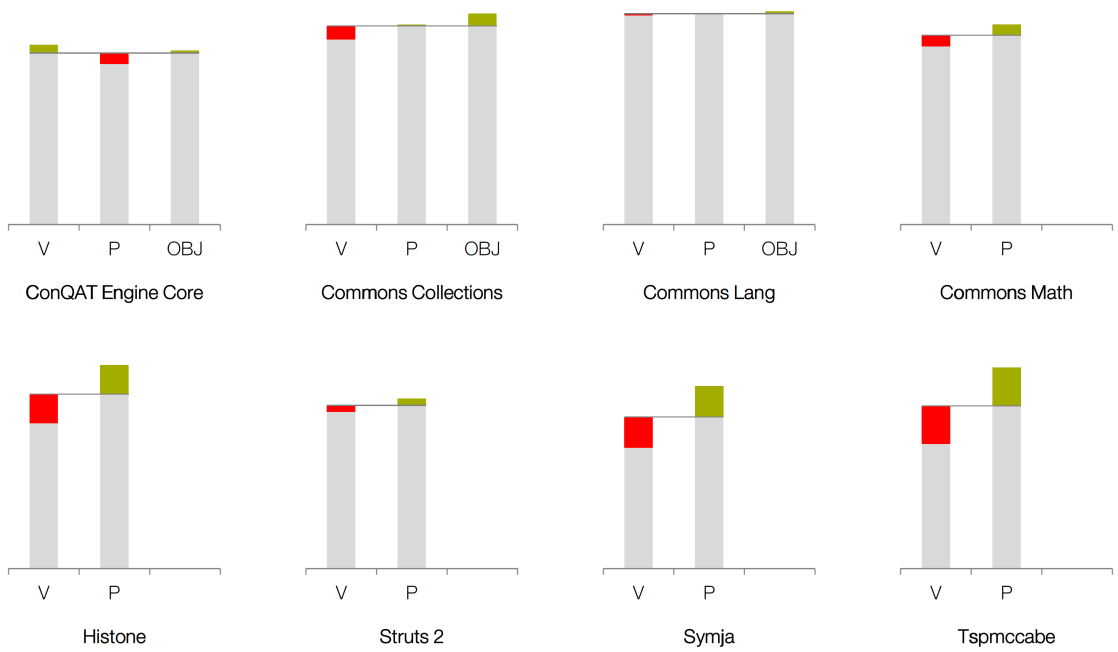


Figure 6.6: Differences in the method mutation score by the return type

compared to the ratio of not inadequately tested methods, evidently higher among the methods with few instructions (about one third higher).

Our findings show that the inadequately tested methods are in general shorter than not inadequately tested methods (true for 6 of 9 projects). Nevertheless, this fact cannot be exploited to find those methods by a static code analysis under the circumstances that most methods are short.

RQ 2.3: Does the mutation testing result of a method depend on its internal coverage?

We presumed the coverage of inadequately tested methods to be significantly lower compared to the other methods. Thus, we expected to find a strong correlation between the coverage and the mutation testing result.

We computed the average instruction and branch coverage of methods of both mutation testing results. Table 6.9 lists the average values and the difference between both testing results separately for each project.
The results suggest that the expectation is appropriate for some projects, but not for all. While the statement coverage of inadequately tested methods in CCSM Commons is on average about 13% lower, projects in which the opposite applies exist, too. The same is the case for the branch coverage.
Thus, this correlation is not given in general. The statement and branch coverage cannot be used as indicators for inadequately tested methods.

Figure 6.8 illustrates the statement coverage of the project CCSM Commons. This project is one which fulfills the expectation. The statement coverage of its inadequately tested methods is significantly lower than the coverage of not inadequately tested ones. This is appropriate for the branch coverage, too.

RQ 2.4: Does the score of a testcase depend on its length?

In order to keep the results statistically valid, we had to exclude further projects with a low number of testcases when answering this and the following questions. Furthermore, we excluded the Histone project in the research questions concerning the relations between the testcase characteristics and the testcase score, because its testcases do not reflect the whole logic. Histone uses an own test runner and defines parts of the testcase logic in JSON files.
We calculated the linear and not linear correlation coefficient (Pearson respectively Spearman coefficient) to determine the relation between the testcase score and the testcase length.

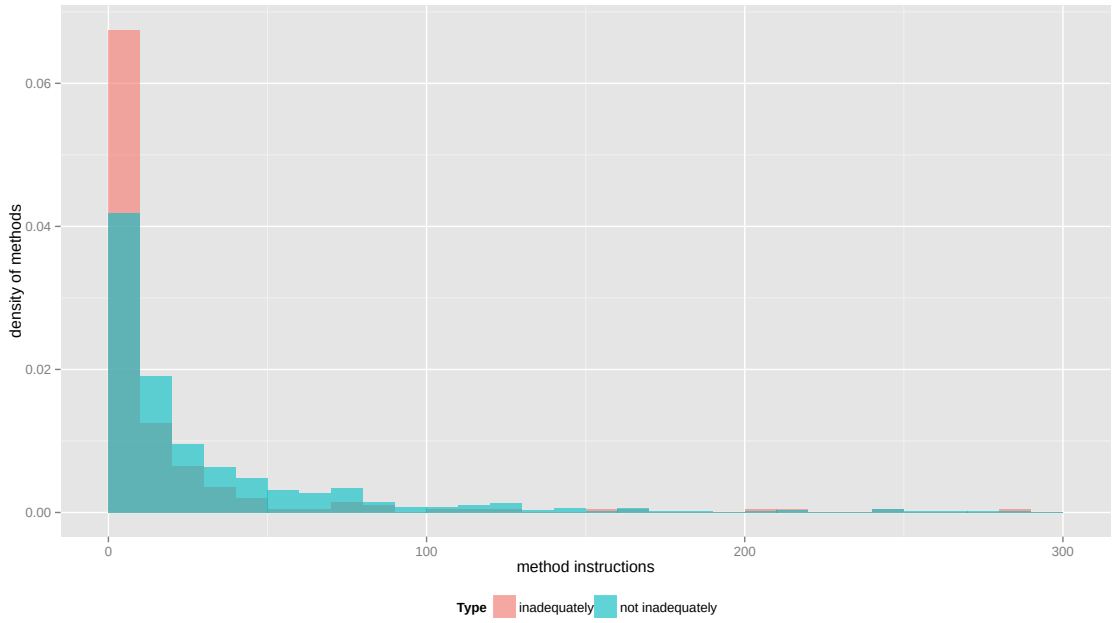


Figure 6.7: Method length of the methods under test (Symja)

Project	Average instruction coverage			Average branch coverage		
	inadeq.	not inadeq.	difference	inadeq.	not inadeq.	difference
Apache Commons Collections	92.86%	95.95%	-3.09%	88.37%	87.43%	+0.94%
Apache Commons Math	98.26%	97.84%	+0.42%	95.24%	92.78%	+2.46%
CCSM Commons	82.15%	95.41%	-13.26%	77.82%	88.80%	-10.98%
ConQAT Engine Core	87.05%	89.72%	-2.67%	83.38%	86.76%	-3.38%
ConQAT dotnet	81.56%	88.03%	-6.47%	66.63%	73.09%	-6.46%
Histone	92.87%	91.15%	+1.72%	78.52%	78.54%	-0.02%
Predictor	97.81%	97.58%	+0.23%	92.11%	88.52%	+3.59%
Struts 2	83.86%	73.71%	+10.15%	47.06%	45.74%	+1.32%
Symja	76.09%	78.31%	-2.22%	80.79%	79.53%	+1.26%

Table 6.9: Average statement coverage of methods

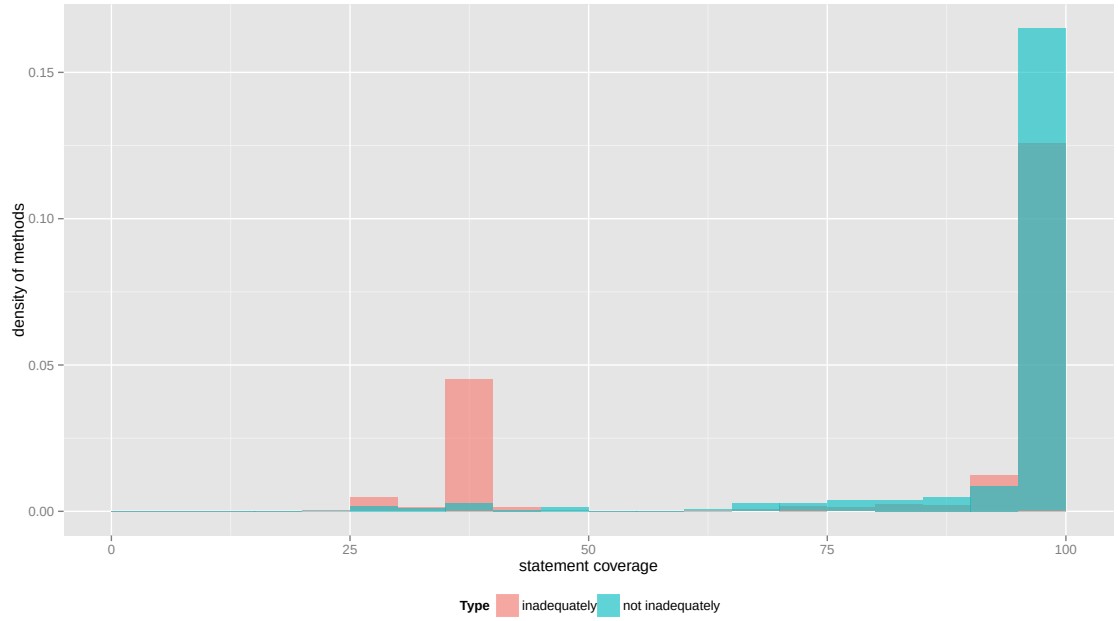


Figure 6.8: Statement coverage (CCSM Commons)

Project	Tests	Score & Length		Score & Assertions		Score & Ass. Dens.	
		Pearson	Spearman	Pearson	Spearman	Pearson	Spearman
Commons Collections	4.142	0.15	0.17	0.15	0.20	0.09	0.09
Commons Math	836	0.09	0.21	0.21	0.28	0.16	0.20
CCSM Commons	417	0.02	-0.06	0.07	0.13	0.14	0.20
ConQAT Engine Core	107	0.28	0.21	0.25	0.30	0.08	0.06
Symja	448	0.08	0.09	0.11	0.10	0.12	0.11
Average		0.12	0.12	0.16	0.20	0.12	0.13

Table 6.10: Correlation coefficients concerning the relations between the testcase score and testcase characteristics

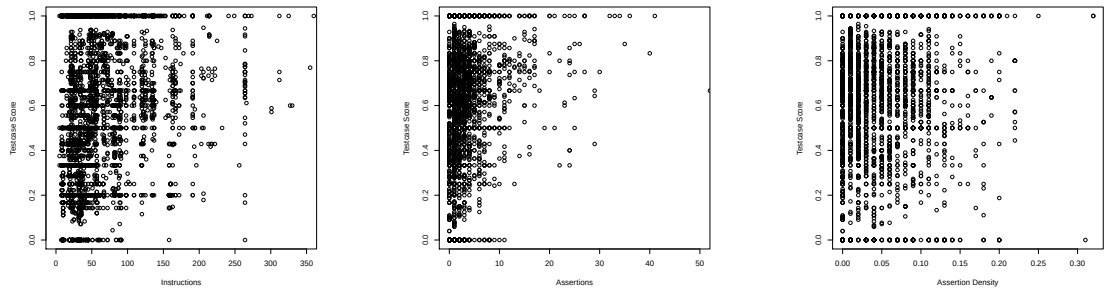
Depending on the project, there is either no or a very low correlation. The Spearman correlation coefficient is between 0.17 and 0.21 for the projects Commons Collections, Commons Math and ConQAT Engine Core, while it is only 0.09 for Symja and even slightly negative with -0.06 for CCSM Commons (see table 6.10). The average Spearman correlation value is 0.12. The linear correlation according to the Pearson coefficient is even lower.
Hence, no reliable correlation between the score and length of a testcase can be proven. Figure 6.9(a) illustrates that for the Commons Collections project.

RQ 2.5: Does the score of a testcase depend on its number of assertions?

The correlation between the number of assertions and the score of a testcase is slightly better than the previous analyzed relation. The average value of the Spearman coefficient for the projects with enough observations is 0.20 and corresponds to a low correlation. It indicates that a testcase with more assertions analyzes its covered methods more reliably.
The calculated coefficients are presented in table 6.10. Figure 6.9(b) shows the scatter-plot for Commons Math. It reaches a positive correlation of 0.28 (Spearman), its linear correlation quality is 0.21 (Pearson).

RQ 2.6: Does the score of a testcase depend on its assertion density?

The assertion density, which expresses how many assertions per instruction a testcase contains, correlates only weakly with the testcase score. The correlation is not existent in ConQAT Engine Core (Spearman: 0.06), while it is low in Commons Math and CCSM Commons (Spearman: both 0.20). The average value of the Spearman coefficient is 0.13 (Pearson: 0.12).



(a) Score and Instructions (Commons Collections)

(b) Score and Assertions (Commons Math)

(c) Score and Assertion Density (Commons Math)

Figure 6.9: Relation between the testcase characteristics and the testcase score

Project	Average number of testcases per method			p-value
	Inadeq. tested	Not inadeq. tested	Difference	
Commons Collections	95.26	23.27	+71.99	< 0.001
Commons Math	8.14	21.89	-13.75	< 0.001
CCSM Commons	4.95	4.90	+0.05	0.961
ConQAT Engine Core	9.00	21.36	-12.36	< 0.001
Histone	27.21	44.07	-16.86	0.013
Symja	5.51	35.51	-30.00	< 0.001

Table 6.11: Average number of testcases covering inadequately respectively not inadequately tested methods and the p-value of the logistic regression

The data of Commons Math is plotted in figure 6.9(c).
Thus, the assumption that the assertion density is an indicator for the test reliability needs to be rejected.

RQ 2.7: Does the mutation testing result of a method depend on the number of tests passed through?

The mutation testing result of a method, i.e. whether a method is inadequately tested or not, seems to be influenced by the number of testcases executing a certain method. We carried out a logistic regression. It states a significant correlation for all projects except CCSM Commons. The significance is denoted by the p-value listed in table 6.11.
Then, we compared the average number of testcases passed through per method separately for inadequately and not inadequately tested methods. These values are also presented in table 6.11. In four of the five projects with a significant correlation, the average number of testcases per method is clearly higher for not inadequately tested methods. (The opposite is the case for the project Commons Collections.) This means that methods covered by many testcases are less likely to be inadequately tested. Though, many methods, which are not inadequately tested and concurrent covered by few testcases, also exist. Thus, it is not possible to detect inadequately tested methods by exploiting this correlation.
Figure 6.10 shows the relation exemplarily for the project Symja.

RQ 2.8: Does the score of a testcase depend on the number of methods it invokes?

The score of a testcase goes along with the number of methods it invokes to a minor degree. We computed the quality of the correlation and obtained an average Spearman coefficient of -0.19 stating a low negative correlation. The coefficient varies greatly between the different projects: The testcases of Histone do not show a correlation, while the coefficient of CCSM Commons is -0.48 (see figure 6.11; note that the dots overlap). That means that 48% of the values of CCSM Commons can be explained by the correlation and suggests that the reliability of a testcase decreases with a rising number of covered methods. This fact is supposed to be one of the reasons, why system tests achieve lower method mutation scores than unit tests.

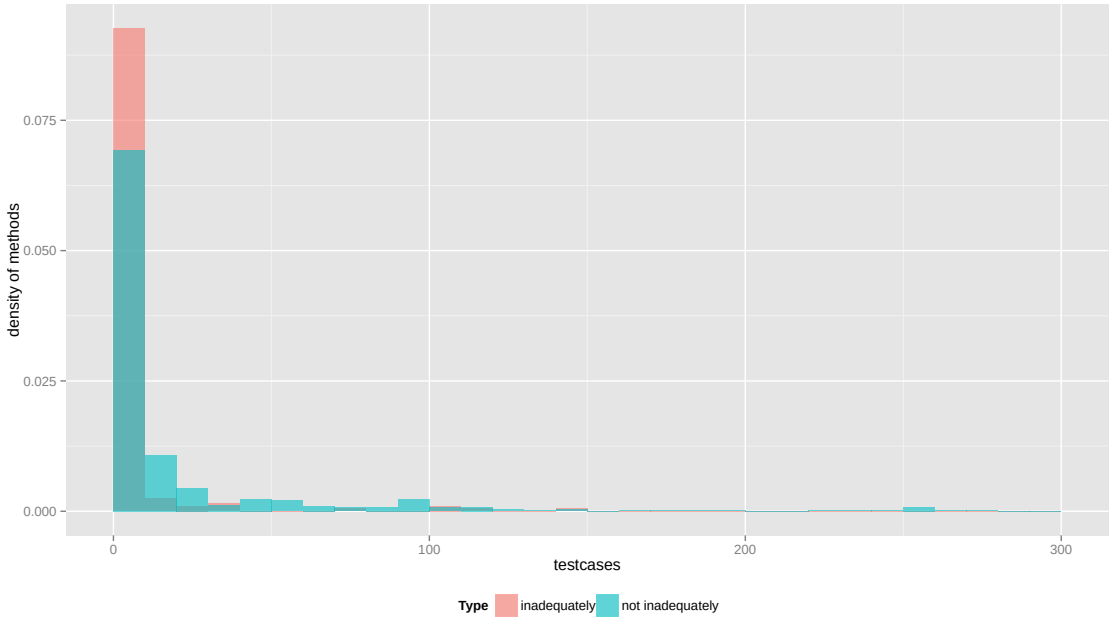


Figure 6.10: Number of testcases covering a method (Symja)

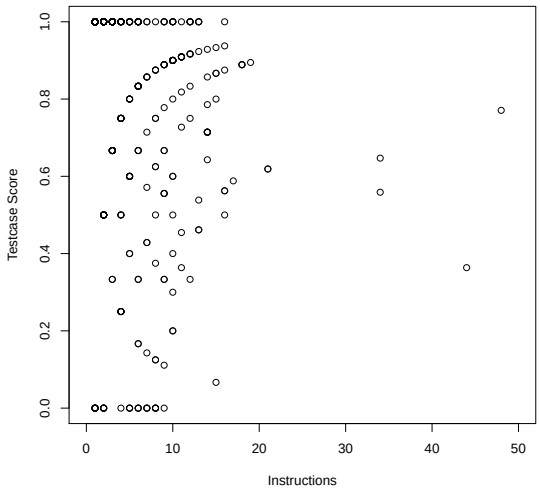


Figure 6.11: Number of methods a testcase covers and its score (CCSM Commons)

6.5.3 Comparison with other Mutation Testing Tools

RQ 3.1: How does the prototype perform compared to other mutation testing tools?

This analysis addresses research question 3.1. We sought to do a comparison between the developed prototype and the mutation testing tools described in section 3.3 (PIT Mutation Testing, Javalanche, MuJava, Judy, Jester).

Applicability
We tried to carry out an analysis for three Apache Commons libraries (Collections, Lang and Net). However, the applicability of the tools was disappointing. Only two of the tools could be configured to analyze at least one of the selected study objects:

- PIT Mutation Testing claims to be the leading mutation testing tool. It kept the promise and managed to generate results for all three study objects.
- Javalanche was applicable only to one project (Commons Net). During the analysis of the two other projects the program either threw undocumented internal exceptions or hung up without progress. The maturity of the program is in doubts.
- MuJava does not enable a fully automated analysis: The first step involves the choice of the mutation operators and the classes to be mutated, the second step executes the tests. While the first step can be done in a single stroke, the second step allows choosing only one testcase per time. Another problem is that some of the generated mutants cannot be compiled. Therefore, it was not reasonable to continue the analysis.
  Given the fact that the program operates on the source code and generates a huge number of mutants, it is considered not to be one of the faster programs. What concerns the results, we expect the inadequately tested methods detected by MuJava to form a superset of the ones detected by the developed prototype. If no testcase is able to detect a mutant created by altering the whole method code (as done by the prototype), no one will in all likelihood manage to detect single mutated lines (as done by MuJava).
- No analysis could be conducted with Judy (in the latest release 2.0 from 2011). Judy requires all testcases to succeed initially. Though, according to the log output, Judy collects all test classes of the project under analysis, but is not able to filter out abstract test classes. Nevertheless, it tries to execute all testcases, but as a matter of course that undertaking causes lots of `InstantiationException`s. Hence, the initial requirement cannot be satisfied and the program is not applicable to the selected projects.
- Jester is no longer maintained and has known problems (see chapter 3.3). We excluded it from the comparison.

Nica, Wotawa and Ramler were investigating in 2011 whether mutation testing was scalable for real word projects. They noted that each mutation tool required different configuration settings and they also experienced major problems when conducting the analysis. Finally, they managed to get one out of three considered tools executed. [41]

The developed prototype supports the analysis of all three study objects. Concerning the general applicability, the restrictions are the following:

- Libraries used by the prototype cannot be used as study objects.[10] Due to the class loading behavior, their modified methods (both during instrumentation and mutation) would not take effect. The used libraries are named in chapter 5.2.
- Some projects have dependencies to other libraries. If one of these libraries is also used by the prototype in another not compatible version, the analysis is not possible. (This concerns especially projects using ASM 3.x as dependency. The prototype uses ASM in the newer but not fully downward compatible version 4.0.)
- The analysis of projects which use own class loaders could be problematic (depending on the implemented behavior of the class loader). It is especially a problem if a project requires certain classes to be only once in the artifacts of the classpath.

Unlike other programs, the prototype tries to be fault-tolerant and is more robust in handling failing testcases. Not all testcases are required to pass initially, failing ones are recorded in the log file and skipped in the further analysis.

10 Special adjustments were performed to allow the use of the prototype as study object (see section 6.4.3).

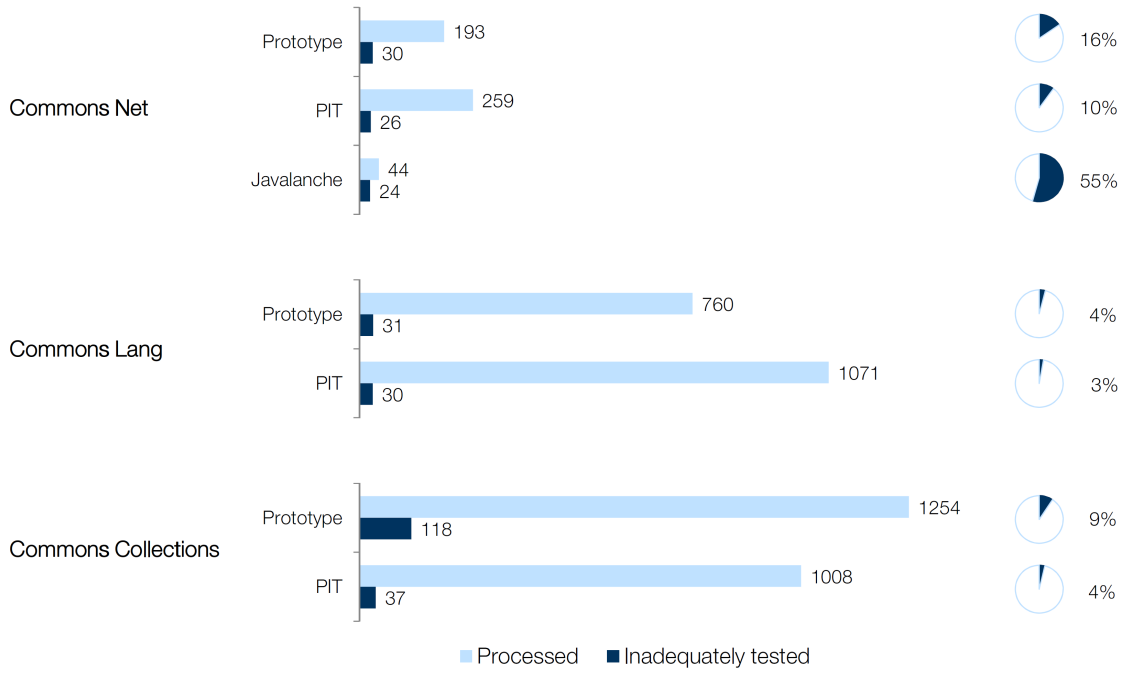


Figure 6.12: Comparison of the number of processed and revealed methods

Results
As described in the study design, we re-executed the analyses with the prototype with settings comparable to the other tools in order to facilitate a more accurate comparison. Thus, the results contain setters and getters within the scope of this and the next research question as well. PIT and Javalanche also include constructors and static initializers in their analyses. We filtered these special methods out before the comparison.

Figure 6.12 illustrates the analysis results of the tools. The light blue bar represents the number of processed methods. The dark blue bar shows thereof the number of detected inadequately tested methods. The resulting ratio of inadequately tested methods is shown at the right side.

Apache Commons Net:
The prototype analyzed methods with void and primitive return types. It processed 193 methods of the Commons Net project, 30 out of these were considered to be inadequately tested. PIT, which generally includes methods returning objects too, analyzed more methods and marked 26 thereof as inadequately tested. Javalanche checked only 44 methods, because it automatically filters methods which it considers to have only a low impact on the program execution. 24 methods were objected.

Apache Commons Lang:
Only the prototype and PIT could be used to analyze this project. PIT analyzed again more methods compared to the prototype. The number of reported inadequately tested methods is with 31 respectively 30 nearly the same.

Apache Commons Collections:
The prototype analyzed about 20% more methods than PIT. PIT also carries out a coverage analysis and applies filters to keep the number of mutants low. This explains, why PIT processed less methods even though it also considers methods returning objects. The prototype listed 118 inadequately tested methods, PIT found 37.

Figure 6.13 refers to the relevance of the revealed methods by considering their severity. The distribution of the results is comparable between the prototype and PIT for all three projects. This is not the case for the Javalanche analysis of Commons Net: About half of the revealed methods are considered to be worthless and not supposed to be tested. They include test helper methods and even testcases.

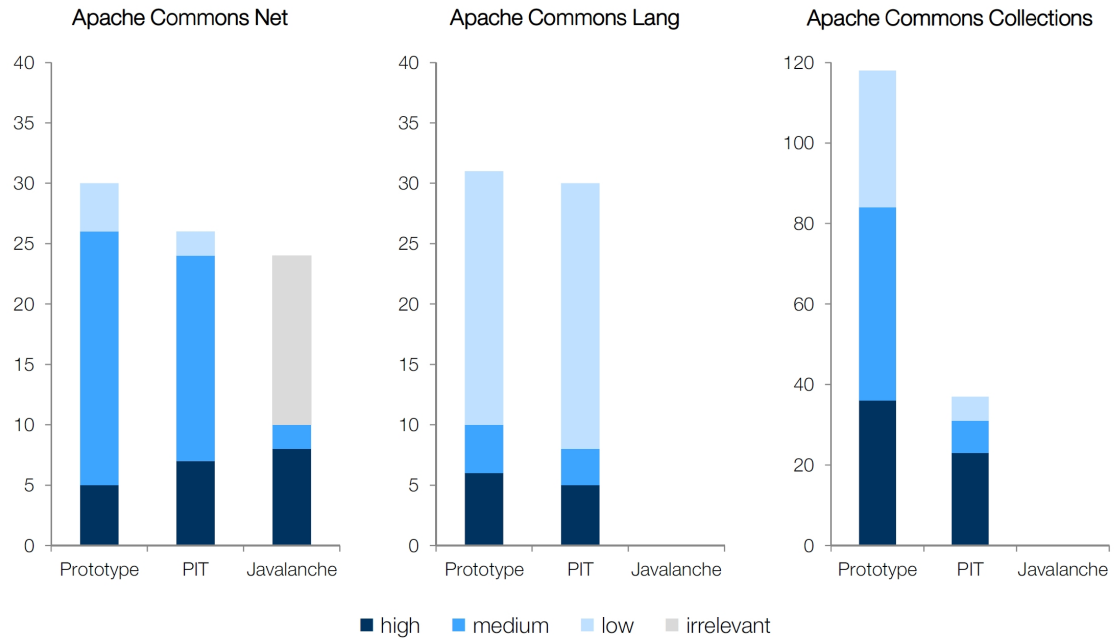


Figure 6.13: Comparison of the severity of the revealed inadequately tested methods

Runtime

Table 6.12 presents the runtime. The prototype needed around 50 minutes for each of the projects.[11] PIT was faster at analyzing two projects, but needed twice as long for the Commons Net project than the prototype. The analysis with Javalanche took less than 8 minutes, but only few methods were considered.

Tool	Commons Collections	Commons Lang	Commons Net
Developed prototype	54:41	44:00	48:28
PIT Mutation Testing	8:38	17:01	93:47
Javalanche	NA	NA	7:48

Table 6.12: Runtime comparison

RQ 3.2: Can the computed results be reasonably combined with the ones of other tools?

Figure 6.14 shows the overlapping of the results of the mutation testing tools. The left side of the figure refers to the overlapping of the processed methods of Commons Net, the right side presents the detected inadequately tested methods of the same project.

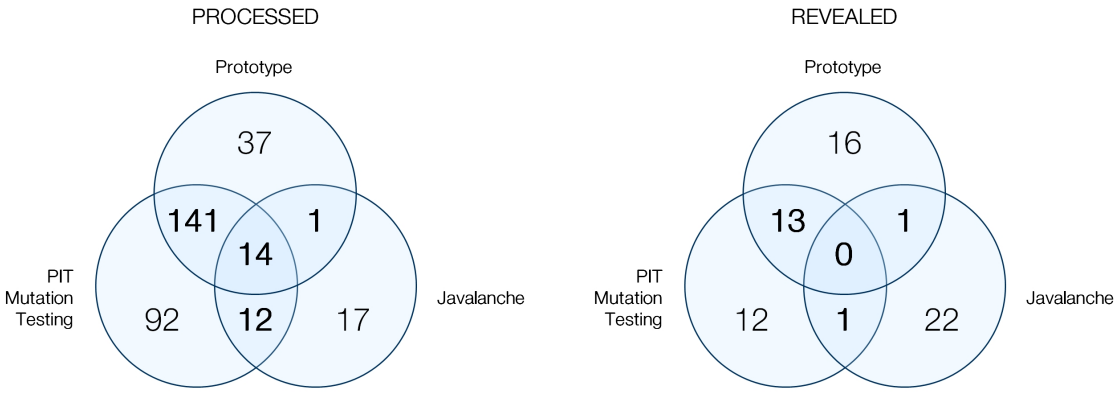


Figure 6.14: Overlapping of the results (exemplary for the project Commons Net)

11 The runtimes of the prototype deviate from the ones listed in the chapter 6.4.4 because of different settings.

One finding is that only 14 methods of the whole project are processed by all tools. PIT processes 141 methods, which overlap with the ones of the prototype. 92 respectively 37 methods will be additionally analyzed if the results of PIT are merged with the ones of the prototype or other way round. This increases as well the number of revealed inadequately tested methods by 12 respectively 16. Javalanche processes further 17 methods and finds 22 methods which are not analyzed respectively revealed by one of the other two tools.
Thus, the combined execution of the prototype and PIT seems reasonable. Such a solution would have processed 270 methods covered by testcases and revealed 41 thereof as inadequately tested. The summed up runtime would be 2 hours and 22 minutes. The number of additionally revealed methods is considered to justify the additional testing effort. One minor issue is the different presentation of the output, but this hurdle is surmountable.

6.6 Threats to Validity

This section presents the threats to the validity.

One threat to the internal validity is that some methods which are considered as inadequately tested might actually be equivalent mutants. We tried to mitigate this issue by the design of the mutation operator and by additional filtering: The used mutation operator modifies the whole method body, while most other known operators mutate only single lines. Such a mutation has a higher impact on the program behavior and is less likely to create an equivalent mutant. Additionally, the generation of two mutants for methods with a simple return type decreases the number of equivalents significantly. Moreover, we filtered methods out which contain no logic (empty methods) or are very simple. These countermeasures reduce the number of equivalent mutants, however, it is likely that some undetected ones remained in the analysis results. At present, it is not possible to automatically detect all of them.

A further threat is that some projects contain testcases which fail on the original code. This can happen because of faulty code (that should actually not be the case in releases) or because of the test environment. Some testcases rely on further data stored in files, on a database with certain settings and a data model or on the network connection. We supplied all the needed and available files in the folder of the test execution and set up databases according to the project manuals. Nevertheless, some testcases still failed. This is the case for example for the Apache Commons Net project, which presumably requires certain firewall settings for some of its testcases. If the excluded testcases had been working, they might have been able to reveal some mutated methods which were not detected by the working testcases. That might influence the result. For this reason, we did not choose any projects which had too many testcases failing on the original code as study objects.

Custom class loaders in study objects could in some very seldom cases affect the results and hence be a threat to the validity. This case occurs, if one class is loaded multiple times by different class loaders during the execution of a single testcase. Otherwise, a custom class loader might indeed load another version of a certain class than the intended modified one, but then that already happens during the instrumentation step. Consequently, not the instrumented version of the class is loaded, thereby its use is not logged and it is not regarded as under test. Accordingly, custom class loaders might cause less classes to be analyzed but do not influence the result in most cases. The same applies if the program restarts itself during the test in a new process with the original jar files. We consider the class loader threat as negligible.

The following threats apply only to some of the research questions:

- Concerning research question 1.3:
  The classification of methods according to their purpose was performed by considering their name. Due to the high number of methods, it was not feasible to inspect the code of all inadequately tested methods to determine their purpose. Some methods might belong to another category and thereby to another severity than the assigned one.
- Concerning research questions 2.4 - 2.6:
  Testcases which source a part of its logic out into another method skew their collected characteristics. The characteristics are the length in instructions and the number of assertions.

The number of assertions might be imprecise to a certain degree due to a further reason: Even though, we were considering project specific assertion classes (if they were broadly used), local verifier methods within a test class were not counted. We also did not take in account assertions specified in annotations such as `@Test(expected = IllegalStateException.class)`.

- Concerning research question 3.1:
  First, some of the analyses which we were not able to carry out with some mutation testing tools for certain study objects might be possible with another program configuration than the tried ones.
  Second, the results of PIT Mutation Testing might contain minor, negligible inconsistencies caused by the incomplete denomination of the mutated methods. The method parameters are suppressed, hence overloaded methods cannot be distinguished.

The external validity concerns the generalization of the results of the case study.

One threat to the external validity is that most analyses were performed only for methods returning void or primitive values including string. The obtained results might not be representative for methods returning instances.

Furthermore, the results of the selected open source projects might not be representative for other projects. We tried to mitigate this issue by considering several projects as study objects. We also paid attention to choose projects of different sizes and application domains.

7 Future Work

This chapter covers open questions which need further research. Furthermore, possible extensions to the approach are presented.

An open question is the ratio of generated mutants which are equivalent. The equivalent mutants are considered as disruptive factors and distort the results. As already mentioned in chapter 6.6, the severity of the mutation operator and the filtering heuristics should keep their number low. Nevertheless, an empirical evaluation should be conducted to analyze the frequency of equivalent mutants in order to confirm this assumption.

A further research question could examine whether the mutation testing result of a method can be indicated by other not yet considered factors (e.g. the cyclomatic complexity of a method) or by a combination of factors (e.g. the product of the method length and its coverage).

One possible extension to the approach is the native mutation support for methods which return objects. Currently, a factory which creates appropriate instances is required to mutate such methods. Some effort is necessary to implement a factory.
In future, objects could be instantiated automatically. This feature can be realized by using the Java reflection API. Hurdles to mitigate are classes which do not provide a constructor without parameters, classes without accessible constructors and return types which represent interfaces or abstract classes. However, these issues are solvable in general: If a class does not provide a parameterless constructor, a constructor with parameters will be used. It will be invoked with default values for primitive argument types and / or with objects which will be built recursively (cycles must be avoided). Not accessible constructors can be made accessible by changing the access modifier via reflection. For return types which are abstract classes or interfaces the class hierarchy can be computed to find instantiable classes which inherit or implement the type.

Another possible extension is the support of further mutation operators.
One promising operator could for example create mutants by removing methods which override a corresponding method in a super classes. Consequently, the method in the super class is executed instead.
In order to allow the use of further mutation operators, the mutation logic should be generalized and support operators provided in plugins.

Moreover, the benefits of an incremental analysis sound promising: The mutation analysis is executed once for the whole project. Subsequent analyses examine only code chunks which were changed (or for which tests were changed) since the last analysis. Thus, an incremental analysis allows a significant reduction of the testing effort. However, its implementation needs to be well though-out and goes along with a loss of accuracy. Unchanged code chunks may depend on other changed classes and make it difficult to decide, whether a certain chunk needs to be re-executed. PIT Mutation Testing already supports this feature.

Finally, the prototype could be extended so that it additionally supports projects of other programming languages than Java.

8 Conclusion

The empirical evaluation conducted in the case study shows that the method mutation score of unit tests is usually in the range of 80% to 95%. It does not deviate heavily among different projects. Thus, the code coverage is not completely misleading and can be used as a rough approximation for the reliability of a unit test suite.
In contrast, the method mutation score of systems is generally lower than the one of libraries. It deviates heavily among the projects and ranges between 30% and 90% for the analyzed study objects. Hence, the code coverage is not a good indicator for the test reliability of system tests. It pretends a reliability which actually cannot be guaranteed by the test suite.

As part of the case study, several correlations were examined. The goal was to find indicators which can point to inadequately tested methods or to weak testcases. We analyzed amongst other things the influence of the length and coverage of methods on their mutation testing result, the length and assertion density of testcases on the test score as well as relations between methods and testcases. The indicators would permit the detection of the desired methods and testcases through a static code analysis. A static analysis would be magnitudes faster than the computationally complex mutation analysis.
However, the results indicate the absence of distinct exploitable indicators. Even though some weak correlations exist, they are not suitable for uncovering the desired elements. Accordingly, there is no way around mutation testing for figuring out the actual reliability of a given test suite.

A couple of tools exists for the mutation testing analysis of Java programs. Some of them are written to meet the needs of academic research and are hardly suitable for real world software projects. In the last years, some new tools emerged which have the intention to be usable in industrial application. They operate directly on the bytecode and thus avoid the comparatively expensive compilation procedure. In addition, they provide further optimizations for accelerating the mutation testing process and reducing the number of equivalent mutants.
The developed prototype also takes steps in this direction and delivers reasonable results in a promising execution time. Further development will be undertaken to extend the prototype by more functionalities (such as new mutation operators) and optimizations.

The benefit of a hybrid solution combining the analysis results of two or more mutation testing tools was examined, too. The findings suggest that the overall result can be improved by such a solution. Especially the combination of the prototype with the solidly working PIT Mutation Testing looks promising. It enlarges the analyzed scope and reveals an increased number of inadequately tested methods.

Bibliography

[1] Martin Fowler. AssertionFreeTesting. `http://martinfowler.com/bliki/AssertionFreeTesting.html`. Visited on September 1st, 2013.

[2] International Organization for Standardization. ISO 26262: Road vehicles – Functional safety, 2011. Cited in `http://www.sebastian-schneider.eu/cms/iso-26262/fault-error-failure-oder-doch-nur-fehler`, September 1st, 2013.

[3] Peter Haggar. Java bytecode: Understanding bytecode makes you a better programmer. `http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`. Visited on September 1st, 2013.

[4] Bill Venners. Bytecode basics: A first look at the bytecodes of the java virtual machine. `http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html`. Visited on September 1st, 2013.

[5] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society, 2007.

[6] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[7] Auri Vincenzi, Márcio Delamaro, Erika Höhn, and José Carlos Maldonado. Functional, control and data flow, and mutation testing: Theory and practice. In *Testing Techniques in Software Engineering*, pages 18–58. Springer, 2010.

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

[9] A Jefferson Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. In *Computer Assurance, 1996. COMPASS'96,'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 224–236. IEEE, 1996.

[10] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[11] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 158–171. ACM, 1996.

[12] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[13] RA DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, 1991.

[14] Torsten Cleff. *Basiswissen Testen von Software: Vorbereitung zum Certified Tester (Foundation Level) nach ISTQB-Standard*. W3l GmbH, 2010.

[15] Lasse Koskela. Introduction to Code Coverage. *JavaRanch Journal*, January 2004, 2004.

[16] Brian Marick. How to misuse code coverage. In *Proceedings of the 16th Interational Conference on Testing Computer Software*, pages 16–18, 1999.

[17] Audris Mockus, Nachiappan Nagappan, and Trung T Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301. IEEE, 2009.

[18] Anna Derezinska. Object-oriented mutation to assess the quality of tests. In *Euromicro Conference, 2003. Proceedings. 29th*, pages 417–420. IEEE, 2003.

[19] Roger T Alexander, James M Bieman, Sudipto Ghosh, and Bixia Ji. Mutation of Java objects. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 341–351. IEEE, 2002.

[20] Upsorn Praphamontripong and Jeff Offutt. Applying mutation testing to web applications. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 132–141. IEEE, 2010.

[21] Pedro Reales Mateo, Macario Polo Usaola, and Jeff Offutt. Mutation at system and functional levels. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 110–119. IEEE, 2010.

[22] Jeremy S Bradbury, James R Cordy, and Juergen Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Mutation Analysis, 2006. Second Workshop on*, pages 11–11. IEEE, 2006.

[23] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[24] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[25] 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.

[26] Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, 2009.

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

[28] Roland H Untch, A Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 139–148. ACM, 1993.

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

[30] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for Java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 297–298. ACM, 2009.

[31] Bernhard JM Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 192–199. IEEE, 2009.

[32] Douglas Baldwin and Frederick Sayward. Heuristics for determining equivalence of program mutations. Technical report, DTIC Document, 1979.

[33] A Jefferson Offutt and W Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[34] Henry Coles. PIT Mutation Testing: Basic Concepts. `http://pitest.org/quickstart/basic_concepts/`. Visited on September 1st, 2013.

[35] Jeff Offutt. MuJava Home Page. `http://cs.gmu.edu/~offutt/mujava/`. Visited on September 1st, 2013.

[36] Lech Madeyski. Mutation testing tool for Java. `http://madeyski.e-informatyka.pl/download/tools/judy/Madeyski13_JudyMutationTestingToolForJava.pdf`. Visited on September 1st, 2013.

[37] Ivan Moore. Jester hints and tips. `http://jester.sourceforge.net/hintsAndTips.html`. Visited on September 1st, 2013.

[38] Eric Bruneton. ASM 4.0. A Java bytecode engineering library. `http://download.forge.objectweb.org/asm/asm4-guide.pdf`, 2011.

[39] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java®Language Specification: Java SE 7 Edition*. Oracle America, Inc., 2013.

[40] Javadoc: Class object. `http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`. Visited on September 1st, 2013.

[41] Simona Nica, Rudolf Ramler, and Franz Wotawa. Is mutation testing scalable for real-world software projects? In *VALID 2011, The Third International Conference on Advances in System Testing and Validation Lifecycle*, pages 40–45, 2011.