Muhammad Awais

ANALYSING THE EFFECTIVENESS OF MUTATION TESTING IN REAL-WORLD SOFTWARE DEVELOPMENT

Technology and Communication

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Muhammad Awais
Title	Analysing the Effectiveness of Mutation Testing in Real-World Software Development
Year	2025
Language	English
Pages	31 + 6 Appendices
Name of Supervisor	Mikael Jakas

Software testing is an important part of the software development lifecycle, ensuring dependability and functionality in complex systems. Traditional testing metrics, such as line and branch coverage, have long been the standard for assessing test effectiveness. However, these methods repeatedly fall short in finding subtle errors that can substantially influence software quality. Mutation testing appears as a favorable alternative, introducing intentional faults (mutants/bugs) in the code to assess the robustness of test suites.

This thesis examines the effectiveness of mutation testing in contrast to traditional testing techniques, aiming on real-world software projects. Using tools like Pitest for Java, mutation testing is used to selected open-source projects to examine its fault detection abilities. The study focuses key insights into the correlation between traditional coverage metrics and mutation scores, detecting strengths and limitations of both methods.

The results determine that mutation testing gives a deeper understanding of test suite quality, revealing flaws often examined by conventional metrics. This research offers practical recommendations for integrating mutation testing into software development workflows and implies areas for future development in testing tools and methodologies.

Keywords Software quality assurance (QA), mutation testing, software testing, fault detection, and software development lifecycle

CONTENTS

ABSTRACT 2
1 INTRODUCTION 5
1.1 Background of Study 5
1.2 Problem Statement and Objectives 6
1.3 Scope and Limitations 6
1.4 Use of AI in This Thesis 7
1.5 Overview of Chapters 7
2 LITERATURE REVIEW 9
2.1 Traditional Testing Metrics 9
2.2 Introduction to Mutation Testing 9
2.3 Mutation Testing in Real-World Software Projects 10
2.4 Effectiveness of Mutation Testing 10
3 METHODOLOGY 12
3.1 Tool Selection and Configuration 12
3.2 Case Study Selection 14
3.3 Selected Projects 14
3.4 Mutation Testing Application 15
3.5 Data Collection and Analysis 16
3.6 Limitations of Methodology 17
4 RESULTS 18
4.1 COVID Alert Service 18
4.2 Calculator App 21
4.3 Pizza Ordering System 22
4.4 Roman Converter 24
4.5 Recommendations for Improvement 26
5 FUTURE GOALS AND CONCLUSION 28
REFERENCE 31
APPENDICES 32

FIGURES

Figure 1. Pit Test Coverage Report of Covid Alert App 19
Figure 2. Package Summary 19
Figure 3. Covid App Active Mutators 20
Figure 4. Examined Tests 20
Figure 5. Pit Test Coverage Report of Calculator App..... 21
Figure 6. Mutations Report 22
Figure 7. Pit Test Coverage Report Pizza Ordering System 23
Figure 8. Test Coverage Report..... 23
Figure 9. Active Mutators and Tests Examined..... 24
Figure 10. Pit Test Coverage Report of Roman Converter App..... 25
Figure 11. Mutators and Tests Summary 25

1 INTRODUCTION

The demand for software testing is rising as the requirement for modern applications are increasing, especially considering the functionality and reliability of them. Traditional methods such as line and branch coverage frequently fail to detect errors because they do not assess how the code behaves. These methods only measure which parts of the code are executed (Jia & Harman, 2011b). Considering further, mutation testing is efficient, and it enables assessing the effectiveness of test suites by providing distinct insight and clarity regarding test quality. This thesis focuses on the use of mutation testing on real life software projects and benchmarks its effectiveness against traditional ones. This study highlights the ability of mutation testing in open-source projects aiming at the tools that do the work such as Pitest. However, it is not the only concern, issues such as tool integration, mutant generation, and comprehension of results are also significant ones. The thesis aims to outline practical solutions for comprehending the results for researchers and developers and incorporating these solutions into previously made systems.

1.1 Background of Study

In software development, software testing is indeed revolutionary and vital to the development of applications along with meeting standards of quality and trustworthiness. Enhancing software complexity proposes complications for conventional testing procedures such as line or branch coverage to uncover underlying defects, which renders the deployed system vulnerable. Mutation testing provides a unique solution in which faults are purposefully implanted into the codebase with the aim of testing suites identifying them. This procedure extends beyond simply assessing the effectiveness of the test. It also identifies areas where traditional measures are insufficient(Vu Nguyen & Madeyski, 2014). The aim of this paper is to show how mutation testing can fill these gaps and

offers comments on its applicability to real-life problems. This research seeks to leverage open-source projects to analyze the pros and cons mutation testing holds from the perspective of improved fault detection and software quality.

1.2 Problem Statement and Objectives

Conventional methods in software testing such as line and branch coverage fail to capture small details resulting in reliability problems in intricate systems. On the other hand, mutation testing adds artificial defects for deeper analysis although it undergoes from limited tool support and poor integration which negatively affects its adoption (Jia & Harman, 2011b). The main purpose of this thesis is to investigate the practical challenges of mutation testing and its effectiveness compared to traditional metrics and at the same time try to address these issues. The aim is to provide steps that can concretely be acted upon to enhance fault identification in actual engagements. The following are the thesis objectives:

I. Evaluate the effectiveness of mutation testing in fault detecting in real-world software projects.
II. Compare mutation testing results with traditional testing metrics like line and branch coverage.
III. Identify practical challenges and limitations of mutation testing in various scenarios.
IV. Provide recommendations for integrating mutation testing into modern software development workflows.

1.3 Scope and Limitations

This research investigates the evaluation of mutation testing on real life software projects particularly on open-source repositories. These open-source tools include Pitest for Java and Pitest for Java, which serve the

purpose of fault injection and test analysis. The boundaries also include determining the relationship between mutation metrics and other standard measures like line and branch coverage. The study is, however, constrained by the lack of sufficient supporting tools, the fragmentation of mutation testing on larger codebases, and the considerable difficulty in the understanding of the meaning behind mutation scores. These limitations may affect the ability of the conclusions drawn to be applied to the problems facing industrial size projects.

1.4 Use of AI in This Thesis

In this thesis, I have used Chat GPT for ideation and Grammarly to correct the grammar of the thesis. Chat GPT also provided the guidelines to install IntelliJ software and libraries that were used to simulate the project. There were some issues regarding the integration of Pitest with IntelliJ, So I took help from Chat GPT by giving queries "How to integrate IntelliJ with Pitest", "Pitest and IntelliJ" etc. I have ensured the authenticity of the contents and respected copyrights. I have not used AI for literature work, I have read research articles and cited them properly in the thesis.

1.5 Overview of Chapters

This thesis is structured as follows:

Chapter 1 introduces the study, outlining the background, importance of reliable software testing, and the limitations of traditional metrics. It also defines mutation testing and sets the foundation for this research. Chapter 2 explores the evolution of software testing, emphasizing traditional metrics and the principles of mutation testing. It includes insights into prior studies, tools, and real-world applications. Chapter 3 describes the tools and processes used to apply mutation testing. It explains the case study selection, data collection techniques, and the approach for

comparing mutation testing with traditional methods. Chapter 4 presents the findings of the research, detailing the effectiveness of mutation testing, challenges encountered, and comparisons with traditional metrics. Key results are supported with charts and tables. Chapter 5 summarizes the contributions of this research, highlights its practical implications, and identifies opportunities for future advancements in mutation testing.

The objective of this study is to show how mutation testing could address the restrictions put in place by the traditional metrics to improve fault identification and overall software quality in actual projects. The study emphasizes the application of open-source tools for practical purposes by focusing on how mutation testing can improve the identification of sophisticated errors that are usually overlooked by conventional techniques. Moreover, this thesis provides insights and practical guidelines on how the mutation testing technique can be incorporated into the development processes already in use. Because of these comparisons and analysis, the research adds to existing literature on software quality assurance. It advocates for the refinement of software testing techniques within highly advanced software systems.

2 LITERATURE REVIEW

Software testing has been a foundation of software development since the origin of programming. Its development shows the increasing complexity of systems and the growing need for reliability and quality assurance. Early methods focused on debugging but modern testing methods include varied approaches such as functional testing, regression testing, and performance testing, to make certain comprehensive evaluation. Test coverage metrics, for example line coverage and branch coverage are normally used to measure the boundary to which code is tested, achieving as a benchmark for test suite effectiveness. As software complexity rises these metrics face restrictions to capture the complexity of defect finding which is most important in real-world scenarios.

2.1 Traditional Testing Metrics

There are two metrics that are used to check the testing quality and those are Line and branch coverage. Line coverage checks the percentage of performed code lines during testing and branch coverage evaluates the execution of all possible branches in conditional statements for example loops and if else statements. Although these metrics offer significant insights, and they sometimes fail to detect complex and logical errors. For example, achieving 100% coverage does not guarantee fault-free software and these metrics only assess the code paths tested not their quality. These errors highlight the need for more robust testing attitudes (Coles et al., 2016).

2.2 Introduction to Mutation Testing

Mutation testing is a kind of white box testing in which testers make small changes to specific parts of source code of an application to make sure that the software test suite can detect changes. This was a method

proposed to reduce the limitations of traditional metrics. Mutation testing involves the introduction of small artificial variations (mutants) into the code to reproduce real-world defects in projects. Test cases are then assessed based on their ability to identify and eliminate these mutants. Mutation testing gives a deeper analysis of test quality by targeting on the strength of test cases rather than simply measuring code functioning. Common types of mutants include statement mutations, operator replacements, and logical inversion. Tools such as Pitest for Java which programs the mutation process and making it more useful and easier for developers (Madeyski & Radyk, 2010).

2.3 Mutation Testing in Real-World Software Projects

In different open-source real-world projects mutation testing has been applied. Case studies show its capability to enhance test suite effectiveness, specifically for projects with complicated codebases. For instance, studies using tools like Pitest highlight how mutation testing can show overlooked faults in libraries, frameworks, and applications. Mutation testing improves test quality by introducing code changes and checking if tests catch them. However, practical acceptance in the industry remains constrained due to challenges in tool integration, runtime performance, and the need for developer capabilities (Domínguez-Jiménez et al., 2011).

2.4 Effectiveness of Mutation Testing

Several studies have recognized the effectiveness of mutation testing in showing faults that traditional metrics overlook. By simulating a variety of fault scenarios, mutation testing appeals to developers to write more comprehensive test cases. For example, empirical studies show that projects with high mutation scores often demonstrate fewer post-deploy-

ment defects. In real-world projects, it’s used selectively for critical systems but faces scalability challenges. Despite its advantages, mutation testing causes challenges such as increased computational cost, complexity in explaining mutation scores, and the generation of equivalent mutants (mutants that do not change program performance) (Petrovic et al., 2021).

The literature review directs on the increasing importance of testing of mutation as a complement to traditional metrics. While line and branch coverage present a baseline for test evaluation, mutation testing presents deeper insights into test suite robustness. Researchers point out the requirement for improved tools and methodologies to address the challenges of scalability and equivalent mutant detection. By filling the gap between theoretical research and modern practices, mutation testing has the potential to update software quality assurance(Coles et al., 2016).

3 METHODOLOGY

The methodology applied in this study is planned to thoroughly value the efficiency of mutation testing in real-world projects. This study seeks to fill the gap among academic research and industrial purpose by joining theoretical concepts with practical implementation. The focus is on applying an advanced mutation testing tool like Pitest on different open-source projects to analyze its effectiveness. This chapter will mainly focus on the selection of tools, the process of applying for mutation testing, and the methods used for collecting the data and analysis. The goal is to support a replicable framework for upcoming studies although focusing on the challenges faced during the research process.

3.1 Tool Selection and Configuration

Mutation testing tools are designed to simulate real-world errors by introducing careful changes (mutants) to the source code. For this study, a tool is chosen based on its acceptance, ease of integration, and comprehensive reporting. Pitest, known as PIT, is an advanced mutation testing tool known for its efficiency and smooth integration with Java projects. It is also known as state-of-the-art mutation testing system for Java and the JVM. Pitest generates mutants/bugs automatically and gives detailed understandings into mutation scores and undetected faults.

This tool was configured for different development environments to support different project requirements as various projects were selected from GitHub. Pitest was integrated into IntelliJ IDEA and Eclipse IDEs for Java projects but then realized that IntelliJ IDEA is best for mutation testing as it is so smooth and easy. The configuration covered setting mutation thresholds, choosing mutation operators, and linking the tool with Maven or Gradle build systems (Vu Nguyen & Madeyski, 2014). Mutation thresholds portray the percentage of mutants that should be

killed for a test suite to be measured efficient. A lower threshold may indicate unsatisfactory test coverage while a higher threshold makes sure that stronger test cases but may lead to excessive test effort. A mutation limit of 80% was chosen to strike a balance between catching faults and maintaining test efficiency. This is just a general way to set up this tool otherwise for every different project. Mutation operators dictate the type of errors introduced into the code. I used arithmetic operator replacement, logical operator, and conditional boundary alterations. They were chosen as they indicate common programming errors and promise a detailed assessment of test quality. Various challenges were faced during the process setup.

One of the challenges that encountered was dependence conflicts. Pitest required specific versions of Java dependencies, which sometimes caused conflicts in existing projects. Another challenge was framework limitations, Pitest is not always fully friendly with large Java frameworks, often needs manual modifications to test paths and configurations. While setting up Pitest, one of the biggest challenges was the dependencies issue between mutation testing tools and existing project libraries. Some versions of Java dependencies needed for Pitest were conflicting with the project setup and because of that the project faced failures while simulation. This was fixed by adjusting the Maven/Gradle configuration files cleaning and building the project again and making sure that compatible dependency versions were used.

Another problem faced was about getting results in a document from Pitest. Usually when the project simulated for mutation testing gives an Index.html file which can be opened to see mutation testing results on different browsers or even on terminal. But when I started my project, it didn’t give any file. After searching I found out that it is needed to name index.html exactly in our dependencies folder like build.gradle.

3.2 Case Study Selection

The following criteria were used for project selection to make sure the findings were suitable for a wide range of software development setups:

1. **Availability of Comprehensive Test Suites**: Complete test suites were important for projects to effectively evaluate the influence of mutation testing.

2. **Diversity of Domains**: Open-source projects were preferred by different industries to ensure the conclusions were not field specific.

3. **Codebase Size and Complexity**: A mix of small, medium, and large codebases was taken to investigate mutation testing's scalability.

4. **Public Accessibility**: All projects were open source to enable replicability of the research.

3.3 Selected Projects

The applications selected for this study involve a mix of standalone desktop applications and web-based systems. The Calculator App is a simple mathematical tool that runs without network dependencies. The COVID Alert Service relates to an external API for real-time data retrieval. The Pizza Ordering System is a full-stack application requiring a backend connection, while the Roman Converter is a lightweight, standalone tool. All applications are open-source, and I had a full approach to their source code, containing backend components where appropriate. The repositories managed for testing were sourced from GitHub and other publicly presented repositories. Below are the links of the projects:

Calculator App Click here

COVID Alert Service Click here

Pizza Ordering System Click here

Roman Converter Click here

Calculator App is a user-friendly application for performing basic arithmetic operations, scientific calculations, and unit conversions with a clean and intuitive interface. COVID Alert Service is a real-time healthcare application providing COVID-19 statistics, safety guidelines, vaccination site information, and alert notifications based on location-specific infection rates. Pizza Ordering System is an online ordering platform for customized pizza selection, real-time order tracking, and secure payment integration, supporting delivery and pickup options. Roman Converter is a simple yet efficient tool for converting numbers between Roman numerals and standard numerical format, designed for educational and historical reference purposes. Each project was thoroughly analyzed to ensure compatibility with the chosen tools and alignment with the research objectives.

3.4 Mutation Testing Application

The mutation testing process began by introducing small, deliberate changes (mutants) to the source code. The modified code (containing mutants) was executed against the existing test suites for each project. The process aimed to "kill" the mutants by detecting the faults introduced. If a mutant was not detected, it was marked as "survived," indicating a potential weakness in the test suite. The following mutation operators were used:

I. **Arithmetic Operator Replacement**: Changing + to -, * to /, etc.

II. **Logical Operator Replacement**: Flipping && to ||, > to <, etc.

III. **Statement Removal**: Deleting specific lines of code to simulate omissions.

IV. **Control Flow Modifications**: Changing if-else statements to invert logic.

Some mutations caused in runtime errors, preceding to application crashes, specifically in the COVID Alert Service, where mutations in API handling produced requests to fail. In contrast, the Calculator App mainly faced logical errors rather than crashes, as its operations are self-autonomous and do not depend on external dependencies.

One remarkable observation was that mutations involving loop conditions in the Pizza Ordering System managed to infinite loops, producing the application to become unresponsive. This highlighted the importance of testing edge cases in iteration-based logic. During test execution, the following challenges were encountered. Mutants that did not alter program behavior but were indistinguishable from the original code. These required manual review. Some test suites failed due to environmental issues or configuration errors. These failures were debugged and resolved before proceeding.

3.5 Data Collection and Analysis

Data collection focused on two metrics mutation scores and Coverage Metrics. The ratio of mutants killed by the test suit indicates the suites robustness. Traditional metrics like line and branch coverage were used for comparison. Mutation scores and coverage data were exported to structured formats (e.g., CSV) for analysis. The data gathered was analysed using techniques such as Statistical Comparisons and visualization. Mutation scores were compared with line and branch coverage to identify correlations and gaps. Bar charts and scatter plots were used to

present findings. For instance, projects with high coverage but low mutation scores highlighted gaps in test case design. Project-Specific Insights, individual project reports were generated to identify patterns and recommend improvements.

3.6 Limitations of Methodology

Pitest occasionally generated equivalent mutants, skewing mutation scores. Pitest lacked support for complex Java frameworks, requiring additional configurations. Larger projects like ShopEase required significant computational resources to test all mutants, limiting the scope of analysis. Mutation scores required careful interpretation, as equivalent mutants and false positives could affect the results. Data collection relied on open-source projects with available test suites, potentially limiting the generalizability of findings to proprietary software. Efforts were made to mitigate these challenges by refining tool configurations, using cloud resources for scalability, and involving domain experts in data validation.

This chapter detailed the methodology adopted for this study, including tool selection, project evaluation, mutation testing application, and data analysis. By applying mutation testing across diverse projects, this research aims to contribute to a deeper understanding of its benefits, challenges, and practical applications in modern software development.

4 RESULTS

To evaluate the efficiency and strength of the test suites established for each project mutation testing was applied using the PIT Mutation Testing Tool, with version adaptations depending on the certain project. The testing process was executed using Gradle, which served as the build automation tool across all projects. The PIT plugin was configured within the respective build.gradle files and mutation tests were initiated through the terminal by executing the command ./gradlew pitest. The "Appendix A" illustrates the plugin method and dependencies for Covid19 app, same methodology is used for other projects as well. Upon execution, the Gradle build completed successfully as indicated by the terminal output message "BUILD SUCCESSFUL" along with the duration of the build process. Following the successful completion of the build, a detailed mutation testing report was automatically generated in HTML format and stored in the directory path build/reports/pitest/[timestamp]/index.html. These reports were then opened and previewed within the integrated development environment (IDE) to analyze line coverage, mutation coverage, and overall test strength. The understandings gained from these reports provided a clear understanding of the completeness and reliability of the test suites fulfilled in each project.

4.1 COVID Alert Service

The COVID Alert Service is a healthcare-oriented application planned to report live COVID-19 statistics. It is a multi-class, modular project, and mutation testing was used to evaluate the efficiency of its wider test coverage. The results showed 94%-line coverage (63 out of 67 lines executed) and 85% mutation coverage (34 out of 40 mutants killed), which indicates that while most of the code was well-tested, there are areas that require improvement. The alert microservice tests are given in "Appendix B" .

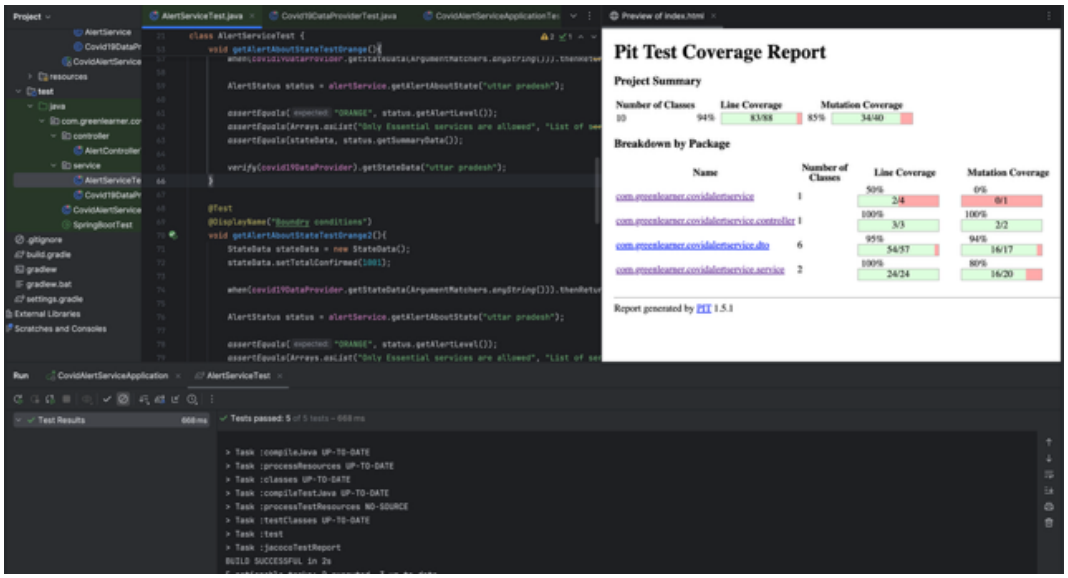


Figure 1. Pit Test Coverage Report of Covid Alert App

Figure 1 shows the project summary, where most packages achieved near-complete coverage, except for the service package, which showed only 16% mutation coverage, revealing a testing gap.

Pit Test Coverage Report

Package Summary

com.greenlearner.covidalertservice.dto

Number of Classes	Line Coverage		Mutation Coverage	
6	95%	54/57	94%	16/17

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
AlertStatus.java	100%	10/10	100%	3/3
BaseDataClass.java	100%	13/13	100%	4/4
CountryData.java	100%	7/7	100%	2/2
CovidApiData.java	70%	7/10	67%	2/3
StateData.java	100%	7/7	100%	2/2
SummaryData.java	100%	10/10	100%	3/3

Report generated by PIT 1.5.1

Figure 2. Package Summary

Figure 2 presents the DTO package summary, where classes like Alert-Status.java, StateData.java, and SummaryData.java achieved 100% coverage, but CovidApiData.java had only 70%-line coverage and 67% mutation coverage due to one surviving mutant.

CovidApiData.java

```
1   package com.greenlearner.covidalertservice.dto;
2
3   import java.time.ZonedDateTime;
4
5   /**
6    * @author - GreenLearner(https://www.youtube.com/c/greenlearner)
7    */
8   public class CovidApiData {
9
10  //     private boolean success;
11
12      private CountryData data;
13
14     private ZonedDateTime lastRefreshed;
15
16      private ZonedDateTime lastOriginUpdate;
17
18  //     public boolean isSuccess() {
19  //         return success;
20  //     }
21  //
22  //     public void setSuccess(boolean success) {
23  //         this.success = success;
24  //     }
25
26      public CountryData getData() {
27 1        return data;
28      }
29
30      public void setData(CountryData data) {
31          this.data = data;
32      }
33
34      public ZonedDateTime getLastRefreshed() {
35 1        return lastRefreshed;
36      }
37
38      public void setLastRefreshed(ZonedDateTime lastRefreshed) {
39          this.lastRefreshed = lastRefreshed;
40      }
41
42      public ZonedDateTime getLastOriginUpdate() {
43 1        return lastOriginUpdate;
44      }
45
46      public void setLastOriginUpdate(ZonedDateTime lastOriginUpdate) {
47          this.lastOriginUpdate = lastOriginUpdate;
48      }
49  }
```

Mutations

```
27  1. replaced return value with null for com/greenlearner/covidalertservice/dto/CovidApiData::getData → KILLED
35  1. replaced return value with null for com/greenlearner/covidalertservice/dto/CovidApiData::getLastRefreshed → KILLED
43  1. replaced return value with null for com/greenlearner/covidalertservice/dto/CovidApiData::getLastOriginUpdate → NO_COVERAGE
```

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- com.greenlearner.covidalertservice.service.Covid19DataProviderTest.[engine:junit-jupiter]/[class:com.greenlearner.covidalertservice.service.Covid19DataProviderTest]/[method:getStateDataTestNoDataFoundForState()] (3 ms)
- com.greenlearner.covidalertservice.service.Covid19DataProviderTest.[engine:junit-jupiter]/[class:com.greenlearner.covidalertservice.service.Covid19DataProviderTest]/[method:getStateDataTest()] (2 ms)
- com.greenlearner.covidalertservice.service.Covid19DataProviderTest.[engine:junit-jupiter]/[class:com.greenlearner.covidalertservice.service.Covid19DataProviderTest]/[method:getSummaryDataTest()] (3 ms)
- com.greenlearner.covidalertservice.CovidAlertServiceApplicationTests.[engine:junit-jupiter]/[class:com.greenlearner.covidalertservice.CovidAlertServiceApplicationTests]/[method:contextLoads()] (13 ms)

Report generated by PIT 1.5.1

Figure 3. Covid App Active Mutators

Figure 3 analyzes CovidApiData.java in detail, showing that a mutation replacing the return value with null in the method getLastOriginUpdate() was not killed. This indicates the absence of a specific test for this method, the microservice test's code is given in "Appendix C".

CovidAlertServiceApplication.java

```
1   package com.greenlearner.covidalertservice;
2
3   import org.springframework.boot.SpringApplication;
4   import org.springframework.boot.autoconfigure.SpringBootApplication;
5   import org.springframework.boot.web.client.RestTemplateBuilder;
6   import org.springframework.context.annotation.Bean;
7   import org.springframework.web.client.RestTemplate;
8
9   @SpringBootApplication
10  public class CovidAlertServiceApplication {
11
12          public static void main(String[] args) {
13                  SpringApplication.run(CovidAlertServiceApplication.class, args);
14          }
15
16          @Bean
17          RestTemplate restTemplate(RestTemplateBuilder builder){
18 1                return builder.build();
19          }
20  }
```

Mutations

```
18  1. replaced return value with null for com/greenlearner/covidalertservice/CovidAlertServiceApplication::restTemplate → SURVIVED
```

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- com.greenlearner.covidalertservice.CovidAlertServiceApplicationTests.[engine:junit-jupiter]/[class:com.greenlearner.covidalertservice.CovidAlertServiceApplicationTests]/[method:contextLoads()] (13 ms)

Report generated by PIT 1.5.1

Figure 4. Examined Tests

Figure 4 highlights another surviving mutant in CovidAlertServiceApplication.java, where a mutation in the rest Template method survived, suggesting the need for direct testing of configuration methods.

Overall, the results reflect a solid foundation, but selective test improvements are needed, especially in untested areas such as service logic and application configuration.

4.2 Calculator App

The Calculator App implements basic arithmetic functionality, specifically focusing on an additional operation. Mutation testing was conducted to assess the reliability and unity of the associated test suite. The project includes a single class, and the mutation testing results showed 100%-line coverage (2 out of 2 lines executed) and 100% mutation coverage (2 out of 2 mutants killed). These metrics confirm that every line of code and every introduced mutation was fully exercised and effectively handled by the tests.

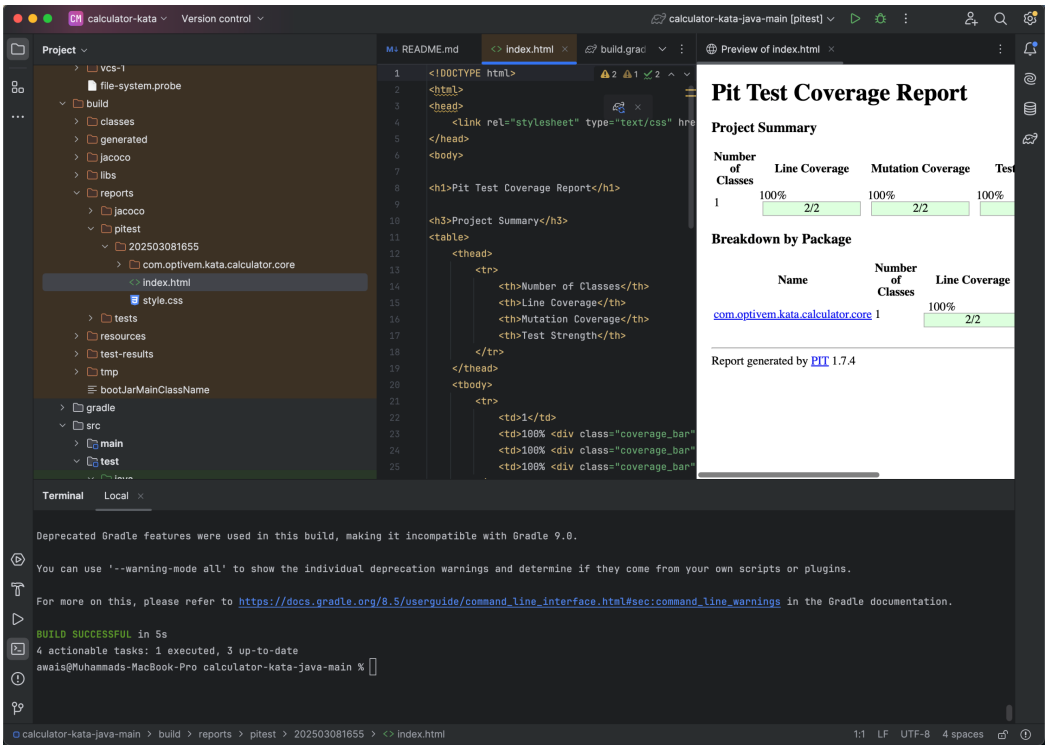


Figure 5. Pit Test Coverage Report of Calculator App

Figure 5 illustrates the PIT Test Coverage Report, where the project summary table indicates full coverage and perfect test strength. The

breakdown by package confirms that the only package involved achieved 100% across all metrics.

Calculator.java

```
1  package com.optivem.kata.calculator.core;
2
3  public class Calculator {
4      public int add(int first, int second) {
5 2        return first + second;
6      }
7  }
```

Mutations

5 1. Replaced integer addition with subtraction → KILLED
2. replaced int return with 0 for com/optivem/kata/calculator/core/Calculator::add → KILLED

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- com.optivem.kata.calculator.core.CalculatorTest.[engine:junit-jupiter]/[class:com.optivem.kata.calculator.core.CalculatorTest]/[method:should_add_two_numbers()] (3 ms)
- com.optivem.kata.calculator.CalculatorKataApplicationTests.[engine:junit-jupiter]/[class:com.optivem.kata.calculator.CalculatorKataApplicationTests]/[method:contextLoads()] (84 ms)

Report generated by PIT 1.7.4

Figure 6. Mutations Report

Figure 6 details the class Calculator.java, highlighting that two mutations were injected: one replacing addition with subtraction, and another replacing the return value with zero. Both mutations were killed, meaning that the existing test case successfully detected and responded to these faults, validating the robustness of the test.

This result indicates that the test suite is comprehensive for the implemented functionality, but as the application expands (e.g., adding subtraction or multiplication), similar testing rigor will be necessary.

4.3 Pizza Ordering System

The Pizza Ordering System handles the full lifecycle of pizza orders, from request building to order storage. Mutation testing yielded perfect results with 100%-line coverage (58/58 lines executed) and 100% mutation coverage (25/25 mutants killed) across six classes. These results confirm a highly robust and complete test suite that effectively covers all logic paths.

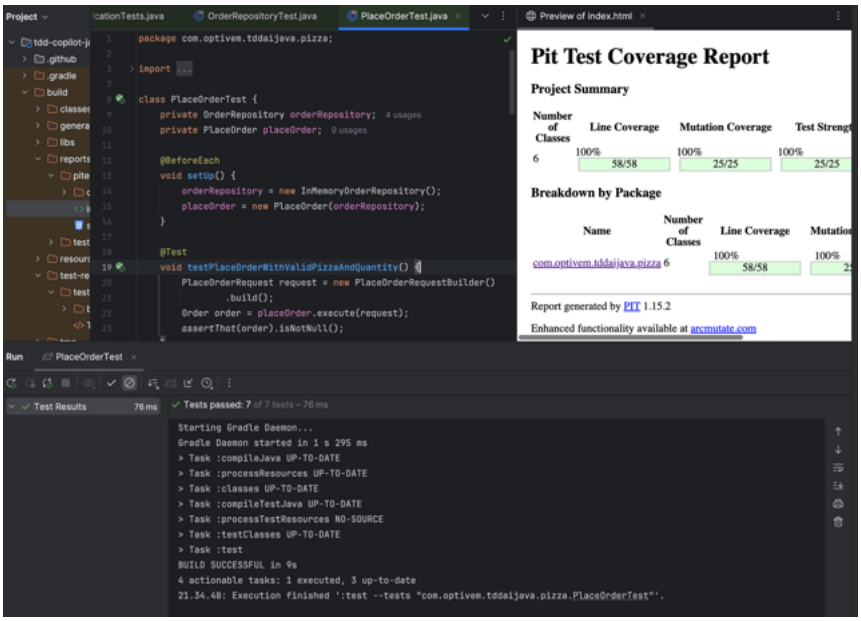
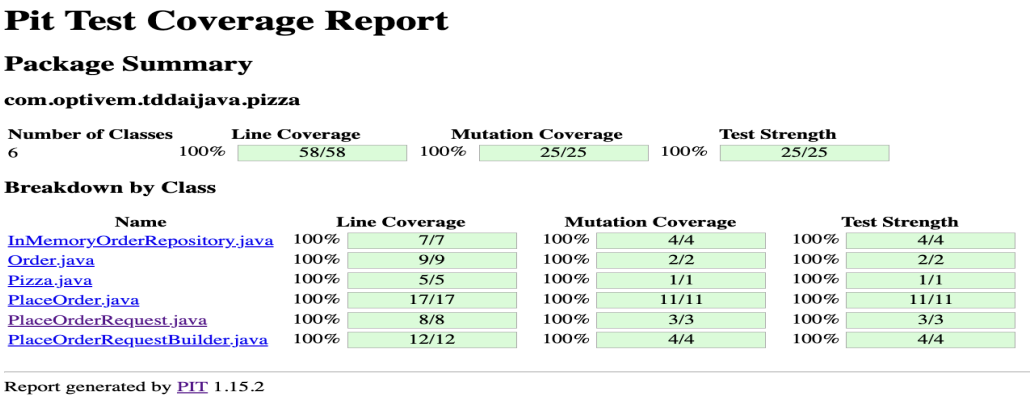


Figure 7. Pit Test Coverage Report Pizza Ordering System

Figure 7 displays the project summary, showing perfect scores in line coverage, mutation coverage, and test strength. Each class, including InMemoryOrderRepository.java, Order.java, and PlaceOrder.java, achieved full coverage, indicated by green bars.



Pit Test Coverage Report

Package Summary

com.optivem.tddaijava.pizza

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
6	100%	58/58	100%	25/25	100%	25/25

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
InMemoryOrderRepository.java	100%	7/7	100%	4/4	100%	4/4
Order.java	100%	9/9	100%	2/2	100%	2/2
Pizza.java	100%	5/5	100%	1/1	100%	1/1
PlaceOrder.java	100%	17/17	100%	11/11	100%	11/11
PlaceOrderRequest.java	100%	8/8	100%	3/3	100%	3/3
PlaceOrderRequestBuilder.java	100%	12/12	100%	4/4	100%	4/4

Report generated by PIT 1.15.2

Figure 8. Test Coverage Report

Figure 8 provides a per-class breakdown, where all lines and all mutations in each class were covered and killed, demonstrating uniform test quality.

InMemoryOrderRepository.java

```
1   package com.optivem.tddaijava.pizza;
2
3   import java.util.HashMap;
4   import java.util.Map;
5
6   public class InMemoryOrderRepository implements OrderRepository {
7       private Map<Integer, Order> orders = new HashMap<>();
8       private int nextId = 1;
9
10      @Override
11      public Order save(Order order) {
12 2        order.setId(nextId++);
13         orders.put(order.getId(), order);
14 1        return order;
15      }
16
17      @Override
18      public Order findById(int id) {
19 1        return orders.get(id);
20      }
21  }
```

Mutations

```
12  1. Replaced integer addition with subtraction → KILLED
    2. removed call to com/optivem/tddaijava/pizza/Order::setId → KILLED
14  1. replaced return value with null for com/optivem/tddaijava/pizza/InMemoryOrderRepository::save → KILLED
19  1. replaced return value with null for com/optivem/tddaijava/pizza/InMemoryOrderRepository::findById → KILLED
```

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- com.optivem.tddaijava.pizza.PlaceOrderTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.PlaceOrderTest]/[method:testPlaceOrderSavesOrderToRepository()] (0 ms)
- com.optivem.tddaijava.pizza.PlaceOrderTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.PlaceOrderTest]/[method:testPlaceOrderReturnsSavedOrder()] (0 ms)
- com.optivem.tddaijava.pizza.PlaceOrderTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.PlaceOrderTest]/[method:testPlaceOrderWithValidPizzaAndQuantity()] (0 ms)
- com.optivem.tddaijava.pizza.PlaceOrderTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.PlaceOrderTest]/[method:testPlaceOrderCalculatesCorrectPrice()] (0 ms)
- com.optivem.tddaijava.pizza.PlaceOrderTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.PlaceOrderTest]/[method:testPlaceOrderAssignsUniqueId()] (1 ms)
- com.optivem.tddaijava.pizza.PlaceOrderTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.PlaceOrderTest]/[method:testPlaceOrderAppliesDiscountForUS()] (2 ms)
- com.optivem.tddaijava.pizza.OrderRepositoryTest.[engine:junit-jupiter]/[class:com.optivem.tddaijava.pizza.OrderRepositoryTest]/[method:testSaveOrderReturnsSavedOrder()] (31 ms)

Report generated by PIT 1.15.2

Figure 9. Active Mutators and Tests Examined

Figure 9 focuses on the mutation analysis of InMemoryOrderRepository.java, where mutations such as replacing addition with subtraction and nullifying return values were all killed. The tests comprehensively verified order saving and retrieval functionality, even under modified or faulty conditions.

The result indicates a very high standard of quality assurance and excellent test coverage, making this test suite a benchmark for future enhancements.

4.4 Roman Converter

The Roman Converter project converts Roman numerals to Arabic integers using logic-intensive processing and validation. Mutation testing confirmed 100%-line coverage (22/22 lines executed) and 100% mutation coverage (19/19 mutants killed), validating the thoroughness and accuracy of the test suite.

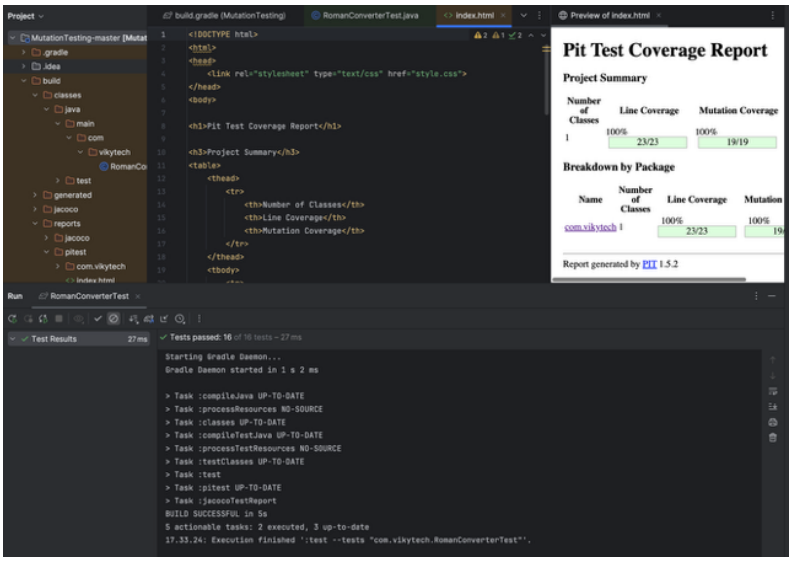


Figure 10. Pit Test Coverage Report of Roman Converter App

Figure 10 summarizes the project's complete coverage, showing that the single class RomanConverter.java was entirely covered and all mutations killed.

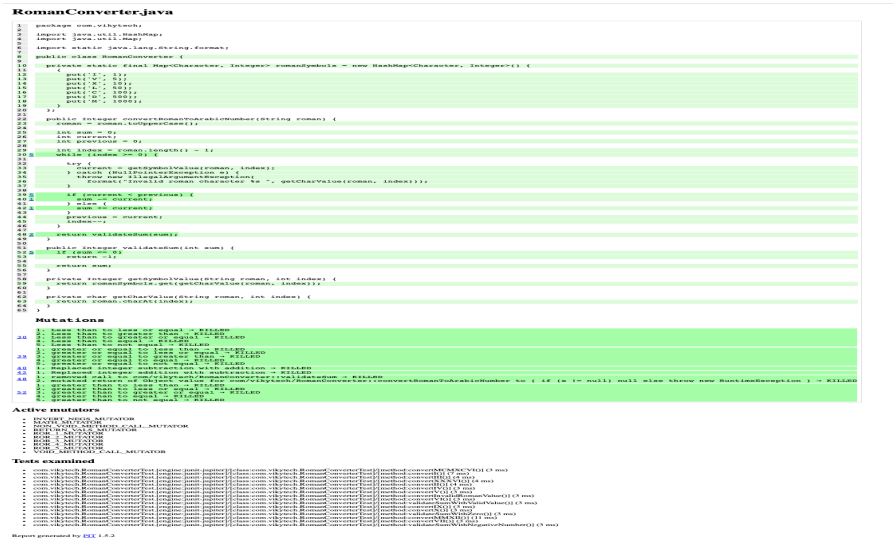


Figure 11. Mutators and Tests Summary

Figure 11 gives an in-depth mutation analysis of RomanConverter.java, where various mutations were injected across logic branches and mathematical operations. These included changing comparison operators, altering arithmetic operations, and returning null or incorrect values. The

test suite effectively killed all mutations, demonstrating that it properly validates both conversion accuracy and input validation. Multiple test cases, including valid conversions (e.g., MCMXCIV, V) and invalid inputs, were examined and confirmed to provide full functional coverage.

This result confirms that the test suite is resilient and complete, covering edge cases and ensuring the correctness of Roman numeral conversion across a wide range of scenarios.

4.5 Recommendations for Improvement

Based on the mutation testing results, various suggestions can enhance the efficiency of the test suites, ensure future scalability, and maintain high software quality across all projects.

1. It is important to expand functional test coverage as new features are added. Applications including arithmetic operations should include tests for subtraction, multiplication, and division, while systems handling orders should test additional features such as discounts, delivery tracking, and order updates. Real-time monitoring systems need larger test coverage for service logic and data handling, and conversion tools should include tests for extended numeral formats and special inputs.

2. Edge case and input validation testing should be strengthened. Tests must account for zero, negative, and large values, as well as invalid or null inputs. This confirms the applications can handle upsetting or boundary scenarios without any failure and improving reliability in real-world conditions.

3. Attention should be given to surviving mutants especially which indicate untested or weakly tested code. Specific methods and packages

with low mutation coverage require additional focused test cases to ensure all logic is properly validated.

4. Automating mutation testing within a CI/CD pipeline using tools like GitHub Actions or Jenkins is recommended. This allows continuous validation of code changes, supports early detection of regressions, and encourages a stable test-driven development process.

5. Keeping build tool compatibility is important for long-term project health. Updating Gradle scripts and plugins ensures compatibility with newer versions, authorizing the use of improved features and avoiding issues related to deprecated configurations.

6. Clear documentation and test clarity should be highlighted. Well-described test cases along with short comments explaining test logic, improve maintainability and team collaboration. Test coverage summaries can also help connect testing scope to sponsors.

7. Lastly, it is appreciated to monitor test strength and performance frequently. Even in projects with full coverage, periodic review ensures that the test suite evolves with the codebase. Identifying and optimizing slow-running tests can also improve development efficiency.

Implementing these recommendations will support ongoing software quality, ensuring that each project remains reliable, scalable, and well-tested as it grows.

5 FUTURE GOALS AND CONCLUSION

The projects discussed in this thesis have showed the importance of strong testing techniques, particularly through mutation testing improving software quality. However, there are chances to further enhance these projects and apply the insights gained to future work. The following goals outline the aim for continued development: Enhanced testing frameworks facilitate expanding the test suites for projects like the COVID Alert Service to address gaps in edge cases and untested application starting scenarios. Integrate automated test case generation tools to ensure complete and consistent coverage during development. Scalability and Performance: Optimize the algorithms used in projects such as the Roman Converter to handle big datasets or more complex input scenarios efficiently. Scale the COVID Alert Service to handle real-time data ingestion and analysis for larger geographic regions.

1. **Real-Time Integration**:
    - Implement real-time features in the Pizza Ordering System, such as live order tracking and dynamic informs.
    - Extend the Calculator App to include more advanced mathematical functions and support integration with scientific applications.
2. **Adoption of Emerging Technologies**:
    - Incorporate machine learning and artificial intelligence into projects like the COVID Alert Service to provide predictive analytics and trend analysis.
    - Explore blockchain-based solutions for enhancing the security and clearness of order management in the Pizza Ordering System.
3. **Cross-Platform Compatibility**:

- Ensure that all applications are compatible with multiple platforms and devices to maximize convenience and user reach.

4. **Comprehensive Reporting**:

   - Extend the mutation testing framework to include more detailed reporting, highlighting not only coverage gaps but also the potential impact of surviving mutations.

By pursuing these goals, projects can continue to evolve, addressing new challenges and leveraging improvements in technology. In conclusion, this thesis has explored the application of mutation testing in evaluating and improving the quality of software test suites across four different projects, Calculator App, COVID Alert Service, Pizza Ordering System, and Roman Converter. Each project determined the importance of robust testing practices and their role in ensuring software reliability, fault tolerance, and maintainability.

The results of mutation testing revealed several key insights:

I. Projects such as the Pizza Ordering System and Roman Converter achieved 100% line and mutation coverage, showing exceptional testing practices.
II. The Calculator App, although relatively simple, demonstrated high-quality testing with complete coverage across all metrics.
III. The COVID Alert Service, while generally well-tested, highlighted areas for improvement, particularly in handling edge cases and application initialization scenarios.

Through the detailed analysis of test coverage and mutation results, this study emphasized the need for continuous evaluation and enhancement of test suites. Robust testing is not only essential for ensuring current functionality but also for maintaining software reliability during future development and scaling.

In conclusion, the insights gained from this thesis support the value of mutation testing as a critical tool in software engineering. By addressing the identified gaps and pursuing the outlined future goals, these projects can serve as benchmarks for quality assurance in software development. Moreover, the methodologies and findings presented in this thesis contribute to the wider understanding of mutation testing and its practical applications, covering the way for its adoption in diverse domains.

REFERENCE

Coles, H., Laurent, T., Henard, C., Papadakis, M., & Ventresque, A. (2016). PIT: A practical mutation testing tool for Java (Demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, 449–452. https://doi.org/10.1145/2931037.2948707

Domínguez-Jiménez, J. J., Estero-Botaro, A., García-Domínguez, A., & Medina-Bulo, I. (2011). Evolutionary mutation testing. *Information and Software Technology*, *53*(10), 1108–1123. https://doi.org/10.1016/j.infsof.2011.03.008

Jia, Y., & Harman, M. (2011a). An analysis and survey of the development of mutation testing. In *IEEE Transactions on Software Engineering* (Vol. 37, Issue 5, pp. 649–678). https://doi.org/10.1109/TSE.2010.62

Jia, Y., & Harman, M. (2011b). An analysis and survey of the development of mutation testing. In *IEEE Transactions on Software Engineering* (Vol. 37, Issue 5, pp. 649–678). https://doi.org/10.1109/TSE.2010.62

Madeyski, L., & Radyk, N. (2010). Judy - A mutation testing tool for Java. *IET Software*, *4*(1), 32–42. https://doi.org/10.1049/iet-sen.2008.0038

Petrovic, G., Ivankovic, M., Fraser, G., & Just, R. (2021). Does mutation testing improve testing practices? *Proceedings - International Conference on Software Engineering*, 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

Vu Nguyen, Q., & Madeyski, L. (2014). Problems of mutation testing and higher order mutation testing. *Advances in Intelligent Systems and Computing*, *282*, 157–172. https://doi.org/10.1007/978-3-319-06569-4_12

APPENDICES

Appendix A

```
plugins                                                                 {
    id    'org.springframework.boot'    version    '2.3.1.RELEASE'
    id  'io.spring.dependency-management'  version  '1.0.9.RELEASE'
    id                                                          'java'
    id                                                       'jacoco'
    id           "org.sonarqube"           version           "3.0"
    id        'info.solidsoft.pitest'        version        '1.5.1'
}

group                       =                       'com.greenlearner'
version                       =                       '0.0.1-SNAPSHOT'
sourceCompatibility                         =                       '1.8'

repositories                                                            {
    mavenCentral()
}

dependencies                                                            {
    implementation  'org.springframework.boot:spring-boot-starter-
web'
    testImplementation('org.springframework.boot:spring-boot-
starter-test')                                                          {
       exclude group: 'org.junit.vintage', module: 'junit-vintage-
engine'
    }
}
test                                                                    {
    useJUnitPlatform()
    finalizedBy                                       jacocoTestReport
}

jacoco{
    toolVersion                          =                       "0.8.5"
//       reportsDir    =    file("$buildDir/customJacocoReportDir")
}

jacocoTestReport                                                        {
    dependsOn test // tests are required to run before generating
the                                                                 report
    reports                                                             {
       xml.enabled                                                   true
       csv.enabled                                                  false
//              html.destination   file("${buildDir}/jacocoHtml")
    }
}

sonarqube                                                               {
    properties                                                          {
       property   "sonar.projectName",   "Covid   Alert   Service"
//              property   "sonar.branch.name",   "test   branch"
       property     "sonar.host.url",     "http://localhost:9000"
//                                  property       "sonar.login",
```

```
"493c84f9bf51f58c9d620d292c4335d1897ed300"
    }
}
pitest                                                          {
    junit5PluginVersion                 =                   '0.12'
    timestampedReports                  =                    false
    outputFormats           =           ['XML',            'HTML']
}
```

Appendix B

```
package com.greenlearner.covidalertservice.service;

import com.greenlearner.covidalertservice.dto.AlertStatus;
import com.greenlearner.covidalertservice.dto.StateData;
import com.greenlearner.covidalertservice.dto.SummaryData;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.mockito.*;

import java.time.ZonedDateTime;
import java.util.Arrays;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;
class AlertServiceTest {

    @InjectMocks
    private AlertService alertService;

    @Mock
    private Covid19DataProvider covid19DataProvider;

    @BeforeEach
    void setup(){
        MockitoAnnotations.initMocks(this);
    }

    @Test
    @DisplayName("When the total number of confirmed cases are
less that 100")
    void getAlertAboutStateTestGreen(){
        StateData stateData = new StateData();
        stateData.setTotalConfirmed(999);

        when(covid19DataProvider.getStateData(ArgumentMatch-
ers.anyString())).thenReturn(stateData);

        AlertStatus status = alertService.getAlertAboutState("Aru-
nachal Pradesh");

        assertEquals("GREEN", status.getAlertLevel());
        assertEquals(Arrays.asList("Everything is Normal !!"),
```

```
status.getMeasuresToBeTaken());
        assertEquals(stateData, status.getSummaryData());

        verify(covid19DataProvider, Mock-
ito.times(1)).getStateData("Arunachal Pradesh");
    }

    @Test
    @DisplayName("When the total number of confirmed cases are
less that 1005")
    void getAlertAboutStateTestOrange(){
        StateData stateData = new StateData();
        stateData.setTotalConfirmed(9999);

        when(covid19DataProvider.getStateData(ArgumentMatch-
ers.anyString())).thenReturn(stateData);

        AlertStatus status = alertService.getAlertAboutState("ut-
tar pradesh");

        assertEquals("ORANGE", status.getAlertLevel());
        assertEquals(Arrays.asList("Only Essential services are
allowed", "List of services that come under essential service"),
status.getMeasuresToBeTaken());
        assertEquals(stateData, status.getSummaryData());

        verify(covid19DataProvider).getStateData("uttar pradesh");
    }

    @Test
    @DisplayName("Boundry conditions")
    void getAlertAboutStateTestOrange2(){
        StateData stateData = new StateData();
        stateData.setTotalConfirmed(1001);

        when(covid19DataProvider.getStateData(ArgumentMatch-
ers.anyString())).thenReturn(stateData);

        AlertStatus status = alertService.getAlertAboutState("ut-
tar pradesh");

        assertEquals("ORANGE", status.getAlertLevel());
        assertEquals(Arrays.asList("Only Essential services are
allowed", "List of services that come under essential service"),
status.getMeasuresToBeTaken());
        assertEquals(stateData, status.getSummaryData());

        verify(covid19DataProvider).getStateData("uttar pradesh");
    }
    @Test
    @DisplayName("When the total number of confirmed cases are
10005")
    void getAlertAboutStateTestRed(){
        StateData stateData = new StateData();
        stateData.setTotalConfirmed(10005);

        when(covid19DataProvider.getStateData(ArgumentMatch-
ers.anyString())).thenReturn(stateData);
```

```
        AlertStatus status = alertService.getAlertA-
boutState("Delhi");

        assertEquals("RED", status.getAlertLevel());
        assertEquals(Arrays.asList("Absolute lock-down", "Only
Medical and food services are allowed here"), status.get-
MeasuresToBeTaken());
        assertEquals(stateData, status.getSummaryData());

        verify(covid19DataProvider).getStateData("Delhi");
    }


    @Test
    @DisplayName("Overall summary test")
    void getOverAllSummaryTest(){
        SummaryData summaryData = new SummaryData();
        summaryData.setUpdateTime(ZonedDateTime.now());
        summaryData.setConfirmedButLocationUnidentified(10);
        summaryData.setConfirmedCasesForeign(1);
        summaryData.setConfirmedCasesIndian(1000);
        summaryData.setDischarged(20);
        summaryData.setDeaths(2);
        summaryData.setTotal(1011);

        when(covid19DataProvider.getSummaryData()).thenReturn(sum-
maryData);

        SummaryData actualSummary = alertService.getOverAllSum-
mary();

        assertEquals(summaryData, actualSummary);

    }
}
```

Appendix C

```
package com.greenlearner.covidalertservice.service;

import com.greenlearner.covidalertservice.dto.CountryData;
import com.greenlearner.covidalertservice.dto.CovidApiData;
import com.greenlearner.covidalertservice.dto.StateData;
import com.greenlearner.covidalertservice.dto.SummaryData;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.web.client.RestTemplate;

import java.time.ZoneId;
import java.time.ZonedDateTime;
```

```
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.anyString;
import static org.mockito.Mockito.when;


 class Covid19DataProviderTest {

    @Mock
    private RestTemplate restTemplate;

    @InjectMocks
    private Covid19DataProvider covid19DataProvider;

    {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    @DisplayName("state data provider test")
    void getStateDataTest() {

        when(restTemplate.getForObject(anyString(), any())).then-
Return(getCovidApiData());

        StateData delhi = covid19DataPro-
vider.getStateData("Delhi");

        assertAll(
                () -> assertEquals("Delhi", delhi.getLoc()),
                () -> assertEquals(4, delhi.getDeaths()),
                () -> assertEquals(0, delhi.getConfirmedCasesFor-
eign()),
                () -> assertEquals(1000, delhi.getConfirmedCas-
esIndian()),
                () -> assertEquals(4, delhi.getDischarged()),
                () -> assertEquals(1000, delhi.getTotalCon-
firmed())
        );
    }

    @Test
    @DisplayName("state data provider test - no data found")
    void getStateDataTestNoDataFoundForState() {

        when(restTemplate.getForObject(anyString(), any())).then-
Return(getCovidApiData());

        StateData maharashtra = covid19DataPro-
vider.getStateData("Maharashtra");

        assertAll(
                () -> assertEquals(null, maharashtra.getLoc()),
                () -> assertEquals(0, maharashtra.getDeaths()),
                () -> assertEquals(0, maharashtra.getConfirmedCas-
esForeign()),
                () -> assertEquals(0, maharashtra.getConfirmedCas-
esIndian()),
```

```
                () -> assertEquals(0, maharashtra.getDis-
charged()),
                () -> assertEquals(0, maharashtra.getTotalCon-
firmed())
        );
    }

    @Test
    @DisplayName("summary data test")
    void getSummaryDataTest() {
        when(restTemplate.getForObject(anyString(), any())).then-
Return(getCovidApiDataForSummary());

        SummaryData data = covid19DataProvider.getSummaryData();

        assertAll(
                () -> assertEquals(5, data.getConfirmedButLoca-
tionUnidentified()),
                () -> assertEquals(100, data.getTotal()),
                () -> assertEquals(2, data.getDeaths()),
                () -> assertEquals(1, data.getDischarged()),
                () -> assertEquals(10, data.getConfirmedCasesFor-
eign()),
                () -> assertEquals(90, data.getConfirmedCasesIn-
dian()),
                () -> assertNotNull(data.getUpdateTime())
        );
    }

    private CovidApiData getCovidApiDataForSummary() {
        CovidApiData covidApiData = new CovidApiData();

        CountryData countryData = new CountryData();
        SummaryData summaryData = new SummaryData();
        summaryData.setTotal(100);
        summaryData.setDeaths(2);
        summaryData.setDischarged(1);
        summaryData.setConfirmedCasesIndian(90);
        summaryData.setConfirmedCasesForeign(10);
        summaryData.setUpdateTime(ZonedDateTime.now());
        summaryData.setConfirmedButLocationUnidentified(5);

        countryData.setSummary(summaryData);

        covidApiData.setData(countryData);
//        covidApiData.setSuccess(true);
        covidApiData.setLastRefreshed(ZonedDateTime.now());

        return covidApiData;
    }

    private CovidApiData getCovidApiData() {
        CovidApiData covidApiData = new CovidApiData();

        CountryData countryData = new CountryData();
       /* SummaryData summaryData = new SummaryData();
        summaryData.setTotal(100);
        summaryData.setDeaths(2);
```

```
        summaryData.setDischarged(1);
        summaryData.setConfirmedCasesIndian(90);
        summaryData.setConfirmedCasesForeign(10);
        summaryData.setUpdateTime(ZonedDateTime.now());

        countryData.setSummary(summaryData);*/

        StateData sd = new StateData();
        sd.setDeaths(4);
        sd.setLoc("Delhi");
        sd.setDischarged(4);
        sd.setConfirmedCasesIndian(1000);
        sd.setConfirmedCasesForeign(0);
        sd.setTotalConfirmed(1000);
        countryData.setRegional(new StateData[]{sd});

        covidApiData.setData(countryData);
/*
        covidApiData.setSuccess(true);
        covidApiData.setLastRefreshed(ZonedDateTime.now());*/
        return covidApiData;
    }
}
```