



Master Thesis

Generating Feedback for Programming Assignments with Mutation Testing

Miftah Adem

Department of Mathematics and Computer Science

Date: 03. February 2023

First Supervisor: Dr.-Ing. Binyam

Second Supervisor: M.Sc. Eyob Sisay

Generating Feedback for Programming Assignments with Mutation Testing

Master Thesis

by

Mifta Adem Ahmed

03 February 2023

Supervisors: Dr. Binyam
M.Sc. Eyob Sisay

Plagiarism Undertaking

I hereby declare that the research contained in this thesis is entirely my original work and that it has not previously been submitted to get any other academic or professional credential.

I certify that the work I have submitted is all mine, with the exception of any pieces that are a part of publications that I have cited. I certify that where references to the work of others have been made in this thesis, appropriate citation has been provided.

Acknowledgement

I want to express my gratitude to Prof. Dr.-Ing. C. Bockisch and M.Sc. Stefan Schulz, who acted as my supervisor and helped me with this project. Secondly, I want to thank my family and friends especially Bede Nnawulezi for their support and enlightening remarks on the study.

I would also like to acknowledge the Computer Science Department at the university of Marburg for their technical support. In addition, I also want to express my sincere gratitude to my loving daughter Seraphim Ify Anumudu, for her great understanding and moral support in completing my Thesis.

Contents

Plagiarism Undertaking	i
Acknowledgement	ii
Abstract	vii
1 Introduction	1
1.1 Overview	1
1.2 Features and Challenges	3
1.3 Problem Statement	3
1.4 Research questions	4
1.5 Motivation	5
2 Literature Review	6
2.1 Introduction	6
2.2 Customized Feedback System	7
2.3 Study on Feedback System and Code Quality	8
2.4 Comparative Study of Mutation Tools and Techniques	9
2.5 Mutation Testing Process	11
2.6 Analysis of the Enhanced Mutation Testing	15
2.7 A research on students' quality-mindedness	16
2.7.1 Diagnostic Evaluation	17
2.7.2 Survey Questions for Students	17
2.8 Feedback systems using mutation testing	18
2.9 A Research on the Impact of Feedback to Programming Students	19
2.10 Conclusion	21
3 Implementation	22
3.1 Project Structure	22
3.1.1 Technologies and Tools used	23
3.1.2 Architecture	25
3.2 Service Flow	27
3.2.1 Main flow	27
3.2.2 Junit test Flow	30
3.2.3 PiTest flow	31
3.2.4 Moving File Flow	33
3.3 Project Diagram	35
3.3.1 Use case diagram	35
3.3.2 Project DirServiceImpl class	36
3.3.3 Sequence diagram	46
3.3.4 Maven library	49

3.3.5	Folders	49
3.3.6	Application properties file	50
3.3.7	Resources	51
3.3.8	Email project tested body.txt	53
3.3.9	Feedback report	57
4	EVALUATION	62
4.1	Overview:	62
4.2	Evaluation Process	62
4.3	Importance of the findings	64
4.4	Research questions:	65
5	Discussion and Future Work	66
5.1	Summary	66
5.2	Interpretations	66
5.3	Advantages	67
5.4	Limitations	67
5.5	Conclusion	68
5.6	Future Work	69

List of Figures

3.1	Project structure	23
3.2	Architecture	27
3.3	Main Flow	29
3.4	Junit Test Flow	31
3.5	PiTest Flow	33
3.6	Moving Flow	34
3.7	Use Case Diagram	35
3.8	Properties Configuration class	36
3.9	Job Start Main Flow class	36
3.10	Project DirServiceImpl class	37
3.11	EmailServiceImpl class	37
3.12	Mutation TestingService class	38
3.13	HtmlReader class	39
3.14	Project DirService Interface	39
3.15	Modify PomXml class	40
3.16	Mutation Sample Message Service class	41
3.17	Mutation Samples File	42
3.18	PiTest Coverage Report class model	43
3.19	Feedback Message Service class	44
3.20	Invalid Project Exception class	45
3.21	Survived Model	45
3.22	Feedback Template Parameter	46
3.23	Mutation Samples Parameter	46
3.24	PiTest report	48
3.25	maven library	49
3.26	Submitted folder configuration	49
3.27	Project Processing folder configuration	50
3.28	Project Success folder configuration	50
3.29	Project Failure folder configuration	50
3.30	Application properties file	51
3.31	Feedback Message file	52
3.32	Mutation Samples file	53
3.33	Email project tested body	54
3.34	Email project_tested_body.txt	55
3.35	Email project_not_pass_unit_test_body.txt	55
3.36	Email project_not_pass_mutation_body.txt	56
3.37	Email project_passed_body.txt	56
3.38	PiTest report location	57
3.39	PiTest report history	58

3.40	PiTest report detail location	58
3.41	PiTest report detail	59
3.42	Feedback report file	60
3.43	A part of Feedback report content	61

Abstract

This thesis describes the development of a fully automated feedback system that was built and made available to offer early learners of object-oriented programming customized and useful feedback hints. To evaluate the robustness of students' JUnit assignment solutions, this system made use of the PiTest mutation-testing tool. Students used this system to submit their assignment solutions and they received customized feedback hints that were better than those generated by PiTest tool.

The study's main objective was to assess the significance and usefulness of customizable feedback hints for early learners taking object-oriented programming courses, as well as the extent to which these hints would be adjusted to provide a successful and productive learning environment.

The findings of this study showed that test-based, fully automated evaluation with the help of the PiTest mutation testing tool could be used to successfully provide a large number of programming students with timely feedback and help them improve their programming skills and knowledge. The technology allows professors or tutors to adjust the feedback system's Mutation Threshold score to determine whether a student's test suite is good or bad at first glance.

Key Words: *Customized feedback system, PiTest, Mutants, mutation testing, JUnit Testing, feedback hints, and test suites.*

1

Introduction

1.1. Overview

Software testing was formerly undervalued because of a variety of issues, including costs and the demand on resources. Software testing has, nevertheless, become more popular recently. As they give students the chance to get real-world experience, programming assignments are crucial to the teaching of computer science.

When a large number of students are engaged in programming assignments, obtaining feedback from them on their work can be difficult for professors to manage. While grades only provide students a basic grasp of how their work compares against the standards, the newly developed customized feedback system that uses a mutation testing approach makes it easier for students to examine test cases and evaluate the quality of these tests automatically.

Programming assignments are essential to the teaching of computer science as they provide students with the opportunity to practice what they have learnt in school and gain practical experience. However, giving students feedback on their programming tasks can be a difficult task for teachers, especially when there are many students involved. On the other hand, grades merely provide students with surface-level insight into how their work compares to the criteria that should be in place for every specific assignment. To address this problem, one possible solution is to utilize mutation testing approach, a method that will examine and evaluate the quality of the tests suite automatically.

Given the importance of feedback in promoting learning and increasing students' knowledge, the development of this feedback system aimed to provide students with meaningful customizable feedback hints to enable them to spot errors in their submitted assignment solutions as well as get suggestions on how to fix those errors.

Students' feedback on their programming assignments can be challenging for professors when a large number of students are engaged. This problem can be solved by the development of an automated feedback system that uses mutation testing process to evaluate tests cases.

In order to fully automate this process, it is necessary to synchronize a Dropbox folder containing student submissions with a local project folder using a program such as Rclone. Additionally, a library such as Quartz can be used to schedule regular scans and executions of the test projects submitted by students. This would ensure that feedback is generated in an automated fashion without any manual intervention required.

The purpose of this study is twofold: (1) explore what type of auto-generated feedback hints are useful for early learners of object oriented programming with basic knowledge of Unit Testing; (2) investigate how we can customize this proposed feedback system to automatically generate meaningful feedback hints for such learners

Tools that help students learn programming have been in use since the 1960s [1]. These tools help students make use of programming knowledge that helps them understand how a program works [2]. Programming is difficult to learn [3] and students need assistance to make progress [4]. This is because tens of thousands of students globally [5] take programming courses, hence teaching and evaluating individual students can be time-consuming and tiresome for professors [6].

Feedback is an essential element of learning [7, 8]. Boud and Molloy defined feedback as "the process by which learners obtain information about their work to recognize the differences and similarities between the standards required for any task given and the qualities of the work itself to produce improved work" [9]. Based on this definition, formative feedback is "information conveyed to the learner with the intention to modify his or her thinking or behavior for the purpose of better learning" [10]. Learners can also gain some insight into their performance through summative feedback in the form of marks or percentages on examinations [11]. On the other hand, grades merely provide students with surface-level insight on their work compared to the criteria that should be in place for every specific assignment.

To evaluate the quality of student-submitted JUnit solutions, a mutation testing process will be carried out with the help of Pitest mutation testing tool. The students will submit their solutions to the provided "projects submitted" folder in a Dropbox and receive meaningful customized feedback hints automatically via their respective email addresses. The professor or tutor should be able to customize the Mutation Threshold score of the feedback system to indicate whether the student test suit is good or bad at first glance. PITest is a powerful tool for generating feedback for programming assignments. It works by introducing small changes (called mutations) to the code and then evaluating whether those changes were detected by the unit tests. If the changes were not detected, it means that the unit test coverage could be improved and we can generate feedback accordingly. In order to customize the feedback message or hints, the feedback system need to read through the mutation testing report that is generated after executing Pitest. This report will contain information about which mutants survived and which were killed by the unit tests. The feedback system will extract this information from the report into a SurvivedMutant model class attribute, which will provide detailed information about each mutation such as its location in the code and any other relevant details. With this information, we can create meaningful feedback messages or hints tailored to each student's programming assignment. These messages can include examples of how to kill certain mutants as well as general tips for improving their unit test coverage. By taking this approach, we can ensure that each student receives valuable feedback on their assignment that is specific to their own codebase which they can use to improve their programming skills.

This thesis will begin by providing a review of related resources on gaining programming knowledge in Section 2. Section 3 details my study topics and methodology while Section 4 presents the evaluation and explores the labelling process. The outcomes are discussed in Section 5, before the paper is concluded with future endeavors outlined in Section 6.

1.2. Features and Challenges

The feedback system is fully automated and the feedback hints are also fully customizable. It is integrated with Dropbox, which is used to store student's projects. It has notification feature to notify the student about testing result via student's email. It also automatically scans student's submitted projects, run JUnit test and send email notification, thereby making it fully automated. Additionally, it also provides to student an example of mutation sample code on how possibly survived mutants can be killed.

The System is designed to move files from a cloud storage service to local storage, which can be configured by time schedule. This enables the system to scan student-submitted projects in the Dropbox folder named "project submitted" and move them to the local storage folder called "projects submitted" for execution. After the projects have been scanned and executed, they will be moved back to the Dropbox folder depending on the test results.

In addition, the system will execute JUnit tests and Pi-tests to verify whether a student's JUnit test cases are passed or not, and compare it with the professor's pre-configured threshold mark. Furthermore, feedback hints will also be provided to students on their JUnit test cases that failed, as well as how they can improve their solutions. Lastly, an email notification will be sent to each student via their provided email address with content based on the JUnit test cases result.

The challenge of providing a means to fully customize the feedback hints for students requires developing an automated feedback system that can be easily customized by professor or tutor. This means creating a means that allows professor or tutor to write custom messages such as what types of feedback they want to give to students and how detailed it should be. In order to fully automate the feedback system, an automated system must be developed that can automatically synchronize Dropbox folder and Local folder using Rclone program. Furthermore, this system needs to be able to automatically scan and execute students submitted test solutions using timer schedule provided by Quartz library.

Providing more helpful and easy-to-understand feedback hints to students requires figuring out a way to allow the professor or tutor to be able to write custom feedback hints for students. Moreover, it should be possible to provide students with code examples in order to support them in improving their test cases. Finally, the PiTest generated feedback needs to be read in order for the professor or tutor's customized feedback hints for students to be effective. This requires designing algorithms that can read through the PiTest auto-generated results and extract relevant information such as which mutants survived or was killed.

1.3. Problem Statement

More than ever, contributions to the field of software development are on the rise. Researchers have previously utilized mutation testing to artificially introduce faults into software to test its robustness. The problem addressed in this research is that many programming assignments do not receive adequate feedback. This is often due to the fact that grading is done manually, and this can be difficult for instructors to provide detailed feedback for every student especially with the fact that a single problem can be described with different algorithms and the same algorithm can be implemented in a number of different ways hence burdening the grading

process. Moreover, automated assessment systems are often limited in their ability to provide meaningful feedback hints to the students. As a result, it is difficult for students to identify and correct their mistakes in order to improve their programming skills.

This research will seek to develop a feedback system that provides meaningful and customizable auto-generated feedback hints to early learners of Object-Oriented Programming (OOP) with basic knowledge of JUnit testing. The proposed system will utilize PiTest a mutation testing tool to evaluate the robustness of the student's submitted assignments. Furthermore, the feedback hint generated by this system will be easy for students to understand, and they will be sent directly to the student's provided E-Mail address.

Finally, this research will explore how useful the customized feedback hints are for early learners while considering how applicable these customized auto-generated feedback hints can enhance early learners' JUnit testing skills.

1.4. Research questions

The contribution to the field of software development has been rising more than ever. Mutation testing is a process of artificially introducing faults into a program and then testing the program to see if the faults are detected or not. If the faults are not detected, then the test suite is said to be not robust and feedback will be generated to the user.

In order to answer the research questions, this thesis will focus on the development of a feedback system that can provide customized auto generated feedback hints to early learners of Object-Oriented Programming (OOP) with basic knowledge of JUnit testing.

This system can be used to evaluate the quality of students' submitted JUnit tests assignment solutions by utilizing mutation testing. The aim is to investigate how useful such customizable feedback hints are for early learners of object oriented programming students with basic knowledge of JUnit testing and how they can enhance their JUnit testing skills.

1. What type of auto-generated feedback hints is useful for early learners of OOP with basic knowledge of Unit Testing?

For early learners of OOP with basic knowledge of JUnit testing, it is very important to provide feedback that encourages exploration and experimentation of JUnit testing. This can be achieved by providing helpful hints about the error in their code, as well as suggestions on how to improve the test cases. Additionally, providing example solutions or snippets of code which demonstrate best practices can help learners understand the concepts more clearly.

2. How can the proposed feedback system get fully automated to generate meaningful feedback hints to early learners of OOP with basic knowledge of JUnit testing?

To fully automate the proposed feedback system, it is necessary to utilize an automated synchronization tool such as Rclone which can synchronize Dropbox folder and Local folder. Additionally, an execution timer schedule provided by Quartz library should be set up so that submitted solutions are automatically scanned and executed at regular intervals. Furthermore, an automated system that makes use of PI-Test should be implemented to generate meaningful feedback hints by simulating errors in the code

and checking if students' tests are able to identify them or not. By combining these components, a fully automated system will be able to provide meaningful feedback hints to students. This will help them learn how to identify bugs in their own programs and write more effective tests.

1.5. Motivation

The process of developing software must include software testing in order to ensure the quality and reliability of the software. Studies have shown that software testing can cost up to 50% of the budget for development. Software tests are used to determine how well developers understand a program's limitations and potential problems as well as how it should operate. Test Driven Development (TDD) is an approach used in the current development cycle with the goal of creating "well known" software that is thoroughly tested in nearly every conceivable aspect. Code coverage is a measure used to track how much of our code is covered in tests, providing information on the written source code's many elements such as conditionals coverage, line coverage, and others.

Despite having high code coverage, we frequently encounter mistakes, bugs, and other problems in production. This raises questions about the effectiveness of our tests; are they sufficient? How many tests do we have? Are all the edge scenarios where our program fails being tested? These questions can be answered through mutation testing. Mutation testing was first proposed over 40 years ago and entails putting changed versions of the source code through software testing. By using mutation operators, modified copies are produced which should fail when tested due to their altered source code. This can help us improve our test suites by better identifying bugs and errors present in our programs.

2

Literature Review

2.1. Introduction

In this chapter, I present a literature review of the various topics related to feedback systems, automated testing tools, mutation testing tools, and other technologies. I aim to identify potential gaps that exist in the application and implementation of feedback systems for early learners of object-oriented programming with basic knowledge of JUnit Testing. Additionally, I will also explore how mutation testing can be used to generate automated feedback hints.

The first paper focuses on a Feedback System that uses metrics and mutation testing to assess and enhance the code's quality. This paper also provides an overview of different types of automated feedback systems such as static analysis, dynamic analysis, requirements-based test generation, rule-based test generation, and formal methods. The paper further outlines the importance of personalized feedback in helping students become more self-directed learners and understand programming concepts better.

The second paper explores the use of metrics and mutation testing in order to measure code quality. It explains that while traditional approaches are good at measuring code complexity or structural coverage, they are not ideal for assessing code quality as they do not measure the correctness or robustness of a program accurately. The paper further introduces mutation testing as an effective way to evaluate code quality by introducing faults or errors into a program and then determining if these errors are detected by a set of test cases.

The third paper compares different mutation tools and techniques that can be used for automated software testing. It discusses how mutation testing can detect bugs that traditional approaches miss due to their focus on code coverage rather than correctness or robustness. Additionally, it provides an overview of different types of mutants such as statement deletion mutants, value assignment mutants, conditionals deletion mutants, etc., as well as techniques for generating these mutants such as classic mutation operators or genetic algorithms.

The fourth paper focuses on the various difficulties software testers encounter when using mutation testing tools such as false positives or difficulty in setting up the environment correctly so that all mutants can be generated accurately. It further provides some tips on how to reduce these issues such as using multiple source files or using random selection when selecting which mutants should be tested first.

The fifth paper looks at the impact enhanced mutation has on software testing performance by comparing it to traditional methods such as statement coverage or branch coverage metrics.

It finds that enhanced mutation is more accurate at finding errors in programs than traditional methods since it considers both structural information about a program and its data flow behavior while traditional methods mainly focus on structural information alone. Additionally, it also finds that enhanced mutation is more efficient than traditional methods since it requires fewer test cases in order to achieve similar results thus reducing the overall cost associated with the software testing process.

The sixth paper focuses on researching students' attitudes toward quality assurance practices such as writing unit tests as part of their programming projects even though they may not receive any points for doing so from their instructors or mentors. The study found that students who were given an incentive for writing unit tests were more likely to write higher quality tests than those who were not incentivized; however, this difference was small compared to those who received no monetary reward at all but still wrote higher quality tests due to having higher motivation levels stemming from their own intrinsic motivation rather than external rewards.

The seventh paper explores how feedback systems can be improved by utilizing mutation testing tools in order to generate automated hints for students when they make mistakes while coding. By providing personalized hints based on each student's individual mistakes made during coding sessions, instructors and mentors can help them correct errors faster thereby improving overall learning outcomes significantly. Furthermore, by providing customized feedback hints, students become more self-directed learners which encourage them to take more responsibility for their own learning process.

The eighth paper looks at how feedback impacts students studying programming courses. The research found that providing timely feedback lead to improved programming skills even if no incentives were offered; however when incentives were offered then there was a significant improvement in student performance compared with those who received no incentives at all. Furthermore, when asked about receiving timely feedback most students felt satisfied with their experience regardless of what incentives were provided with some even recommending that teachers provide more detailed comments about their work so that they could better understand where they made mistakes.

2.2. Customized Feedback System

The literature review is a chapter that will assess previous and existing research studies related to feedback systems, the application of the PiTest tool, mutation testing tools, etc. Many developers have come up with feedback systems to help lecturers provide automated feedback to their students. The development of automated feedback systems that are dependable, stable, and give meaningful feedback to beginner students can be a challenging task; however, increased research on trending technology has enabled developers to overcome this challenge. Providing customized feedback can help students improve their coding skills and understand programming concepts better [12]. Sharing and exchange of research knowledge have become crucial as new researchers keep emerging with new knowledge. By providing feedback that is customized to each student's code, instructors and mentors can provide guidance that is more detailed to students, helping them to identify areas where they need to improve and correct any mistakes they may have made. Furthermore, by giving students customized feedback, we can help them become more self-directed learners, thereby encouraging them to

take more responsibility for their own learning process.

Fraser and Zeller made several assessments while exploring tools used for generating test cases from mutations and unit tests from mutants [13]. These tools such as the PiTest allow programmers to generate mutants more efficiently and in a faster way [14]. PiTest can be quite useful in the testing process because it is very efficient and fast compared to other tools because it operates on bytecode and optimizes mutant executions in comparison with other tools. [15]. This can help save time as well as improve the feedback by simplifying it in a more easy way to understand for students especially early learners with basic knowledge of JUnit Testing.

Researchers have also explored how mutation testing can be used to generate automated feedback that is customized to each student's code [16]. This type of mutational analysis-based feedback system has been found to provide greater insight into a student's code than traditional approaches such as static analysis or manual inspection. By utilizing mutation testing, instructors can quickly analyse a student's code and pinpoint key issues and errors made while writing JUnit Test. Mutation testing has proved an important process that has been used in this thesis to test the quality of the source codes.

This literature review will identify potential gaps that exist in the application and implementation of customized feedback systems for beginner students that have basic knowledge of JUnit Testing. More so, the literature review will also explore and try to draw similarities that exist between mutation testing and traditional approaches such as static analysis or manual inspection.

2.3. Study on Feedback System and Code Quality

To study feedback systems and code quality, the concept of mutation analysis is considered [17]. It is cost-inefficient to use lesser mutation operators because they make a downside at the reliability factor. Therefore, to overcome these challenges, the incremental subsets of deletion operators are utilized which is possible at a lower price and higher reliability.

Many programs are developed in different Java languages and frequently involve numerous hardware gadgets and software modules. In order to deal with such complications while making maintenance responsibilities easy, inventors document programs with code comments and design documents. Code comments that are well-written aid developer understanding, problem-finding, and maintenance. On the other hand, the syntax of code comments is not enforced by the grammar of a programming language or examined by its compiler and is written in phrases of the English language.

Static analysis tools and linters offer only minimal syntactic support for verifying the validity of comments. Thus, developers are primarily responsible for producing comments of high-quality and making sure they are up in projects. Researchers have paid a considerable amount of attention over the past decade to the issue of rating the quality of the comments in code. Researchers have a stake in this area, but they cannot seem to settle on a single definition of quality when discussing comments in code [18]. It is not easy to provide a blanket definition of quality when discussing code comments due to the wide variety of contexts in which they may be used.

Reviewing one another's work is a useful engineering practice since it guarantees high-quality, maintainable code, and facilitates knowledge sharing within a team. However, the

worth and advantages teams gain through code review depend on the quality and utility of the comments they receive. Coding reviews involve multiple programmers looking at a programmer's most recent changes and discussing any issues they may have found [19]. Finding bugs and ensuring the code is of high quality are the primary objectives of a code review. Keeping those two aims in mind is essential, even though code reviews give a far larger set of benefits, such as information distribution, learning, and mentorship.

The biggest problem with code reviews is that they slow down development time for some teams. So, the team's output suffers from the time spent performing code reviews. Unit testing has been a standard practice for quite some time, and as teams expand their capabilities, the size of their test suites grows [20]. Executing tests that span several components or involve integration takes more time. Since all code needs tests, and more of them, these suites will expand even further with the advent of unit testing techniques like TDD. A high number of unit tests can be a solid testing foundation. Still, they can significantly increase test execution time, especially if those tests are later extended to cover integration or components [21]. Knowing the precise effect of each code change, the tests that must be executed, and the potential requirement for brand-new tests is crucial for deciding what to test.

Finding problems earlier in the software development lifecycle is preferable and less expensive than finding them later when they might cause significant delays in the project's timeline. Many times, developers do not run enough or any tests because they are unsure which ones to run.

As the build is configured to execute the whole suite of tests, the development team must wait for feedback/validation from the build process before proceeding with their work. Instead, development teams can use test impact analysis to determine which tests must be executed before code is committed into a build to validate the changes. The CI process can provide faster feedback to developers when code changes result in failed tests thanks to test impact analysis. It is the ideal scenario for development managers to make sure their teams perform tests before the code is checked in, but in reality, this rarely occurs. Furthermore, they want to ensure that their teams are informed as soon as possible once the code is checked in as to whether or not it caused any test failures. As a result, test effect analysis must extend to the CI process and the developer's workstation.

2.4. Comparative Study of Mutation Tools and Techniques

A paper by [22] has a detailed study of the JUnit in several mutation testing programs. This paper evaluated multiple mutation testing challenges that occurred to the software testers. The foremost challenge considered in this paper was the rising computational expenditure while executing the mutation testing. Although there are various other advantages of mutation testing, which are unfortunately overpowered by the disadvantage of computational expense.

This paper further introduces approaches reduce the cost while performing mutation testing process by incorporating multiple tools such as automating the process in Java. After reviewing multiple papers related to mutation testing, the authors were able to interpret a word cloud and word frequency plot using the R-programming language by implementing these features into their analysis. It was observed that the software testers frequently utilized the mutation and test. To consider the most effective testing software, one can be evaluated using the mutation score which interprets the outcome of the mutation testing.

In addition to these, the authors also studied the two approaches to generate the mutants which are the source code and byte code. With the JUnit, various other mutation testing programs for Java were also evaluated. The known mutation testing programs are MuJava, Jester, Javalanche, Jumble, PITEST, and Judy. Each of these programs has its own merits and demerits. To compare the program's in-detail process, the paper here suggested a set of algorithms that could show the effectiveness of the program. The authors here precisely described the merits and demerits of programs in each test case. In the bottom line, it was concluded that PITEST, Jumble, and Javalanche were the most effective mutation testing programs with automation. Furthermore, the authors also stated the future scope of work, which aimed to integrate the cloud framework.

Since mutation testing is a complicated process, it is necessary to automate this process to simplify it by utilizing a convenient and easy-to-use software solution that facilitates the process [23]. The proposed approach aims to overcome these challenges. For this approach, the paper has incorporated the point cut and advice mechanism which is based on oriented programming. This mechanism accelerates the process and avoids complications.

The paper also studied and described various other studies. The authors also contrasted other Java mutation testing tools such as Response Injection (RI), Judy, Jumble, MuJava, and Jumble. In the comparison, it was noted that Judy has a unique and distinct characteristic from the other mutation testing tools. It was also observed that MuJava has certain barriers which hinder the smooth process of mutant generation and compilation phases.

The novel method overcomes various programming fallacies such as crosscutting concerns. Generally, in mutation testing, the compilation phase is the most time-consuming stage. Therefore, with the implementation of the Fast Aspect-Oriented Mutation Testing Algorithm (FAMTA) light, the compilation phase can be enhanced although this method brings some alterations to the mutant generation process.

During the generation process, the group of mutants is considered rather than the single mutants. This group of mutants is managed by meta-mutant which are responsible during every FAMTA light testing phase. This paper also described various mutation operators supported by Judy. Although, during the implementation stage, the authors stated various issues occurred. Due to the exorbitant memory size, the program caused fallacies such as performance and reliability challenges. In the conclusion, it was observed that FAMTA light could eliminate the multiple iterations which ultimately enhanced the generation and compilation stages.

In [24] a comparative study on mutation testing techniques was done. The techniques for which the software testers look are cost efficiency and feasibility. The mutation testing technique is among the most complex to use, yet it is also one of the most reliable. Therefore, in this paper, the authors compared and evaluated the different mutation techniques. Random sampling method-level, class-level, and all operators are the commonly used mutation techniques.

About five Java applications are created and assessed to test each strategy. Initially, a certain algorithm is considered which is then followed for all the test case techniques. In MuJava, class-level operators are further broken down into inheritance, encapsulation, java-specific features, and polymorphism. The syntax prescribed for this operator is inserting, deleting, and modifying the expressions. Then the method-level operator which is a conventional operator in MuJava based on procedural language features follows. Like class-level operators,

it also has some syntax prescribed such as inserting, replacing, and deleting.

The operators are further divided into assignment operators, shift operators, conditional operators, arithmetic operators, logical operators, and relational operators. The article used these methods in addition to the automated mutation tool to improve the outcome. Five Java applications named coffee-maker, black-jack, cruise control, elevator, and find were implemented on each of the four operators. In the final experiments, the conclusion should be that all operators sampling the most effective operator. This method could effectively eliminate the mutants and provide a corroborative output of the test case. It was observed that the effectiveness of the operator is proportional to the detection of the mutants. For the future scope of work, the author aims to study in-depth relations of applications in terms of operators and applications.

The sequential trend of each development phase is described and presented in [25]. Then the theory of mutation testing is discussed. The coupling effect and the competent programmer hypothesis are the key essential hypotheses. The process and problems of each hypothesis were also discussed. Much of the cost that is spent on computing makes mutation testing expensive. The mutation-reducing strategies, including mutant sampling, weak mutation, parallel, mutant clustering, SIMD, selective mutation, high order mutation, firm mutation, compiler, mutant schemata, MIMD, and interpreter, should be taken into consideration in order to minimize costs.

In [26], the author also discussed the development phase of each technique. On the other hand, comes the execution cost reduction technique, which is a strong, weak, firm mutation, runtime optimization technique consisting of interpreter-based and compiler-based. Numerous programming languages, including Java, Fortran, C, AOP, C, web services, security policies, SQL, network protocols, FSM, and state charts, have been subjected to mutation testing. Various other techniques and tools were also discussed with the implementations and applications. In addition to these, the historical implementations, usage, and studies related to this were also discussed.

There were some unresolved issues, obstacles, and expanses of success with the study. Equivalent mutants have been one of the unresolved problems in the studies. In addition to this, mutation testing is a highly expensive method which is a great barrier to implementation although it was seen that there were many points of convenience due to the larger implementation of applications and the effectiveness of the modal. The author states that for the future scope of work, there shall be increased in-depth research and study on the semantic effect.

2.5. Mutation Testing Process

The concept of test-driven development had been popular in the development phase of the software although the idea of mutation testing has been seen to be more successful than the former. To explore the hybrid model using both test-driven development (TDD) and mutation testing, the author of this research recommended. This study being novel was first introduced in this paper. The novel technique was named as TDD-M approach. The primary motive of this study was to obtain a comparison result between single and hybrid techniques [27]. The author of the paper also discussed multiple research papers. According to the TDD approach, the developer first creates the test case before writing the model code. Although

the developer must create a sizable test case, the benefit of this situation is that certain minor codes loop.

Unlike TDD, mutation testing is implemented after writing the code. Certain parts of the codes are iterated with minor alterations for the mutants. The mutation score in this case determines the code's result. The white-box or fault-based testing technique was another term for the principle of mutation testing. Instead of evaluating the test cases, the author of this work analyzed the hybrid model using agile programming techniques.

Confronting confirmation biases is more likely when the TDD technique is used, and as a result, it may be quickly removed. With the hybrid model of TDD-M, the bias could be effectively reduced when evaluating the test case directly. During the experimentation of TDD-M, TDD, and the Mutation test, the most effective and reliable approach was the hybrid model, TDD-M. Although during the implementation, some fallacies could be eliminated with further studies.

Furthermore, the number of open-source applications containing test cases was investigated. Moreover, the applications' use of various frameworks was taken into account. To sort the project with the word, the paper here considered six different datasets with the results for the further process. Upon these different parameters were implemented such as file filtering and search method which were implemented in either java and Kotlin files or project files. Once the programs were sorted, each of these programs was deeply examined to understand the reason behind the occurrences. The module of random selection implemented the process of deep examination. The article here then investigated the relationship between the word's frequency and the project properties using scatter plots. Each program's settings were put into practice using the Pearson coefficient relation.

Further detailed graphical representation of the number of occurrences is provided in the research paper. In the bottom line, it was observed that about 51.57% of all projects consisted of test cases. In addition to this, it was also observed that the utilization of certain tools to predict the relevance of the project was not that productive. In the analysis of the usage of framework, the most used framework denoted is JUnit4/JUnit5, Mockito, spring framework, hamcrest, etc although, there was a certain threat to the validity. For the future scope of work, the author aimed to study the example test and its automation for detection. In addition to this, the author aimed to enhance the tool by making it more time-efficient and convenient.

Similarly, [28] analyzed the metric suites for the JUnit test code. The primary aim was to propose a unified metric suite. The testing effort's development phase was interpreted and documented. Five-unit test cases were used in the study by the author. They include TNOO, TINVOK, TDATA, TLOC, and TASSERT.

Six different open-source java programs, which are ANT, JFREECHART, JODA-Time, Apache Lucene Code, POI, and IVY were considered to proceed further. Considering different parameters, these programs were chosen. After the detailed interpretation of each parameter, it was observed that POI was the largest program in terms of classes whereas the smallest program in terms of classes is the JODA. Although in terms of the line of codes, the JFC is the largest. In addition to this, it is also observed that the JFC program is the most covered for the JUnit classes.

To understand the proposed metric suites, the module of Principal Component Analysis (PCA) was implemented. This module also analyses the internal dimensions of the program. There is a link in between the metrics for unit test cases and the internal software class

characteristic. Additionally, the study used XLSTAT to carry out the analysis in order to put this into practice. Although it poses an internal threat, it is possible to identify the connections between Java classes. On the other hand, the external threat of validity showed that the JUnit test is only developed for certain classes. In the final evaluation, the author successfully identified the test case metrics. It was also observed that TLOC and TINVOK had enhanced the obtained pieces of information. With the addition of CodePro, the author hopes to generate JUnit test cases automatically for the future scope of work.

Nguyen, Quang, and Madeyski [29] conducted a thorough analysis of the various strategies for software test case prioritization which include Test Suite Minimization and Regression Test Selection. This model generally implements the historical applications, which are also cost-time-aware, and requirement risk-aware. The paper aimed to resolve certain research questions as well. Here, the metrics utilized for the test case prioritization and synonymously studied topic were analyzed. The coverage awareness technique's objectives were to increase the test case's comprehensive coverage of all test case elements and to reveal more about the programming language.

Historical awareness processes the statistical data. In this process, cost reduction and control are the primary motives, which is why the cost-cognizant approach comes into place. During the implementation phase, the time of the implementation is also a concern that is overcome by the time-aware approach. The requirement and risk-oriented approach adhere to the least available requirement and improved effort. With the growing requirements and demands, the source code must be altered periodically. This can cause inefficiency and can also introduce various bugs. To overcome this, the model-based approach is implemented whereas the approach for GUI/Web applications emphasizes the current time requirements.

Apart from these, there are also various other approaches for real-world applications. In the final evaluations, it was observed that nowadays the Test-Driven Development (TDD) had been gaining momentum. Through this, the efficacy of the program had been increasing thereby reducing the overall expenditure. The author provided a certain conclusion with the sets of future scope of work.

A powerful method for evaluating the efficacy of test suites is mutation testing. To determine how many mutations are eliminated, mutants are created and tested alongside the test suite. Mutation testing is therefore generally acknowledged a computationally costly method. Mutation testing commonly referred to as "program mutation," has been extensively studied over the past 40 years but rarely used as a testing criterion.

Mutation operators in code produce variants of a program called mutants that can be used to mimic bugs or lead testers to edge situations [30]. Testers must find or develop tests that cause these mutations to act differently from the original, un-mutated software in order for them to pass. If a test case fails it is said to have killed but if it passes it is said to have survived. Testers to evaluate and improve test sets that currently exist or to assist in the creation of high-quality tests can use mutation.

Experimental evidence has shown that mutation testing more preferred than data-flow-based testing or control-flow-based. It has also been used to evaluate other test needs, such as test reliability. Despite its success, it is costly and challenging to utilize in practice due to various issues, including the need for several tests, the number of comparable mutations, and the vast sets of mutants that must be conducted, sometimes repeatedly. There could be hundreds of variants generated with just a few dozen lines of code in a straightforward routine.

In contrast, certain mutations present greater difficulties and require extensive research on the part of the tester. Some mutants can be readily and quickly eliminated without any effort (slain by numerous tests). It is not surprising that the hardest-to-kill mutants often make the best test subjects. Nevertheless, it is hard to know if a mutant is similar or just hard to kill (and, in fact, undecidable in general). These costs have made mutation testing impractical for most people. For instance, one possible explanation for the relative rarity of mutation in practice is that it is difficult to locate individuals with the desired mutation. Researchers have responded by developing several cost-cutting solutions with varying priorities. The number of mutants must be decreased, specific mutants must not be produced, mutant execution must be sped up, test set generation must be automated, test sets must be minimized or prioritized, and automatically equivalent mutants must be identified.

Many methods to lower the price of mutation testing have been put forth, created, and researched in the past. According to empirical data, mutation testing is applied to gauge test reliability and other test requirements [31]. Despite its success, it is costly and challenging to utilize in practice due to various issues, including the requirement for several tests, the number of comparable mutations, and the vast sets of mutants that must be conducted, sometimes repeatedly. There could be hundreds of variants generated with just a few dozen lines of code in a straightforward routine. Some mutants, however, present greater difficulties and require extensive research on the part of the tester. Some mutants can be readily and quickly eliminated without any effort (slain by numerous tests).

It is impossible to distinguish if a mutant is tough or hard to kill. Naturally, hard-to-kill mutants frequently serve as beneficial test subjects. The application of mutation testing in practice has been severely limited by these costs. For instance, some contend that the difficulty in locating mutants that are similar to one another is the reason mutation is commonly used [32]. In response, researchers have developed a number of other ways to save money. The reduction of the overall mutant population, the prevention of the generation of particular mutants, the acceleration of the execution of mutants, the automatic formation of test sets, the minimization or prioritization of test sets, and the automatic identification of identical mutants are some more focused objectives.

The fault injection testing technique's most well-known criterion is mutation testing. It assesses a test case set's quality and makes improvements. So, little syntactic changes are made to the program being tested, and a changed copy of the program, known as a mutant, is produced for each change done. Modifications stand in for any errors that programmers might make while writing a program. This criterion encourages the creation of a test case collection that shows the errors introduced in mutants are not present in the original program, increasing the program's dependability [33]. The original program uses the first set of test cases, which used to develop and run mutations.

Those considered dead and no longer employed in the test behave differently from the original software. Analyzing the collection of mutants that are still alive, analogous mutants are found. It is regarded as equivalent when a mutant exhibits the same behavior as the program being tested in all test scenarios. The mutants that are still alive are finally killed in new test situations. Despite the effectiveness of mutation testing, numerous issues remain, including the vast number of mutants produced, the high computational costs associated with their execution, and the significant effort required to identify similar mutants.

2.6. Analysis of the Enhanced Mutation Testing

In a report, Bashir assessed MuJava, a method for assessing evolutionary mutations in Java applications [34]. Mutation testing being highly effective but very expensive needs to be made economically feasible. Therefore, by incorporating multiple approaches with different parameters and advantages, the fallacies of the testing could be eliminated. Implementation of the novel techniques such as genetic algorithm approaches can enhance the mode of testing. The primary motive of this approach is to evaluate the outcome after the implementation and consider the feasibility.

Generally, mutation testing introduces the fault in the programs and generates an efficient test case to overcome the fallacies. In the paper, the author has presented the evolution mutation tools for Java for automation with a certain set of parameters to adhere to. This approach is known as eMuJava which is implemented upon four types of methods. The proposed approach by the paper allows integrating the testing with the multiple levels of the classes. eMuJava works with the 10 operators which provide operations to each mutant with one fault. The partial operators are conventional, and the other parts are object-oriented.

When compared to other testing tools like Offutt, Ma and Kwon have more operators than eMuJava. The article has taken into account four different automated test case generation methods: random testing, standard genetic algorithms, improved genetic algorithms, and genetic algorithms with improved fitness functions. Here, Random Testing generated the test case in the random occurrences where the Standard Genetic Algorithm also initially follows the Random Testing but in a later stage, it learns to overcome the fallacies with the help of the previous iterations whereas the third algorithm is the addition of the fitness function to the standard function. The fourth algorithm is the enhanced counterpart of the third algorithm with the addition of a novel approach to the crossover method.

In addition to this, eMuJava supports various configurative options for feasibility and convenience to perform the testing. There are six configurations, which are mostly utilized. Moreover, eMuJava is a GUI-based tool which is also one of the advantages. The author of this paper has made this program open source by making the source available to all. However, this software has a complicated internal architecture made up of three programs: a test case generator, a code instrumentation program that uses the compiler, and a mutant generator. The author depicts an elaborative graphical representation of the architecture of the program [35].

The eMuJava does certain descriptive processes such as mutant generation, population generation, fitness evaluation, crossover, biological mutation, etc. For the statistical analysis, the paper here implemented the normality test. In the bottom line, this approach was proposed to counter the computational cost and provide convenience to the testers. For the future scope of work, the paper proposed includes more types of evolutionary techniques such as artificial immune systems, particle swarm optimization, and more.

With the increasing, line of codes in large software programs, the complexity of mutation testing is hereby increased proportionally. With the surging computational time, the computational cost of the testing also increases. Therefore, it is generally seen that most entities eliminate the phase of testing in the development protocol. Mutation testing has been an efficacious method to test the given program, although it brings some alterations to the final source which can be very tedious for large software. Therefore, through this paper, the author aimed to reduce the financial burden and study the influence on the source code by

the custom mutation operators.

Considering the two goals, the paper is divided into two sub-parts with a detailed discussion on each. The author of the paper considered Maven as the mutation system integration tool, where the procedure is initially implemented and further studied. The Maven project created generally relied on the Project Object Model (POM) modules. Certain built-in lifecycles are default, clean, and site. In addition to this, there is various other built-in function such as validate, compile, test, package, verify, install, package, etc.

To implement and authorize vendor-independent source control management (SCM) operations, there are a set of tools which is known as Maven SCM. Furthermore, for the framework, the paper considered the utilization of the Java Collections Framework (JCF). In addition, because of its increased efficacy, the installation of mutation testing in the PiTest extension is also taken into consideration. To pace the efficacy of the mutation testing, here the concept of incremental analysis has been implemented. In addition to this, the concept of differential analysis is also evaluated in the paper.

To have a wider overview, the author considered multiple machines for the evaluations. With the primary motive, the author evaluated multiple strategies for cost reduction. These strategies as discussed were incremental analysis, differential analysis, and operations operators. The first method utilizes the caching mechanism to iterate the consecutive mutation tests although when compared with the differential analysis, this method could conveniently shift between multiple program versions without hindering the iterations. Furthermore, the novel operators on mutations had been able to enhance the mutation score of the testing.

In one paper, [36] proposed a method to reduce mutation testing in the java classes. The goal of the paper was to shorten the computing time without compromising the final product by lowering the number of mutations in the program. Furthermore, the paper considered selective reduction. The mutation of the model was considered using the model transformation method. In the later stage, the final mutation is converted to the text form.

Fang suggested utilizing concolic and mutation testing for automated JUnit production and quality evaluation [37]. The general motive of this concept of concurrent provides the simultaneous computational capacity of multiple programs. The proposed mutation tools in the paper are evaluated and experimented with in two systems which are the Banking system and the Incremental system. To create a fault-free program, the exceptional handling capacity of the software must be highly efficacious and reliable. The paper evaluated and tried the multiple instances to check the reliability of the proposed model.

2.7. A research on students' quality-mindedness

Researchers [38] evaluated the quality characteristics of feedback systems, focusing on readability, convention adherence, documentation, and adequate testing. Early programming courses at colleges sometimes fail to address software quality since they concentrate on fundamental topics and frequently only employ brief exercises. Students may not understand the importance of creating high-quality code due to the tiny quantity of the required code and the fact that it is typically not improved upon after submission. In the second research, a survey was created to allow students in upper semesters to rank the significance of several quality factors. This was also utilized to learn more about their programming experience

prior to attending university.

Their findings were divided into a number of categories. The key finding was that success in an early programming course is strongly positively correlated with having a solid understanding of software quality. The study also found that even when they work in teams, students already place a high value on software quality. However, there is definitely room for improvement in quality awareness. Additionally, research has indicated that university classes are where the majority of students receive their first formal training in programming.

2.7.1. Diagnostic Evaluation

The authors were interested in how early learners' ultimate grades may be affected by their knowledge of software quality [38]. A problematic tree-based Perrinsequence implementation that is comparable to the student's prior knowledge of the Fibonacci sequence was given to them for their first exam. Citizens who encountered issues with the Perrin sequence implementation were enrolled for the repeat test. The first exam attracted around 130 candidates, while the second exam attracted roughly 60 students. The other students chose to immediately participate in the repeat exam because 60 of them had previously taken the first exam and failed it. An outcome is a number between -1 and +1, with +1 being the strongest association and 0 denoting no correlation at all. This implies that the highest scores would go to students who had no notion of quality while the lowest scores would go to students who had the notion of quality.

The first exam's results were calculated using correlations, and the final scores were 0.55 (writing tests), 0.41 (spotting/fixing a fault), and 0.61 (entire work) while any number greater than 0.4 was regarded as a favorable connection. The item-test correlation for authoring the tests may still be determined to be 0.34 with a lower sample size of just roughly 60 learners. The majority of test-takers who took the exam again neglected to include the component that contains, resulting in an item-test connection of 0.01 and a task connection of 0.25. The first exam's strong correlation shows that future investigation into the relationship between quality-mindedness and causality is worthwhile.

2.7.2. Survey Questions for Students

The participants in a course that was offered between the second and third semesters were also surveyed for the study. The survey's goal was to find out how much thought students put to software quality while working in groups or alone. Going into every measure would have revealed the survey and possibly discouraged students from taking part. The researchers, therefore, included 9 measures in their poll to avoid this.

[38] Chose the least common denominator of the courses that students were supposed to finish. The idea of Magic Numbers and the "Don't Repeat Yourself" maxim, which seem to be crucial to understanding the metric Duplicated Code, was required of the pupils. The metrics' names were taken into account as the second selection criterion. Students would be able to get some idea of what they were about from those with names that were self-explanatory. Another aspect to take into account while selecting a measure was testing. Both beginning courses and testing metrics address source code testing, which is crucial for assuring the competence of software.

There is little indication that any one group of students stands out in the study, but only 40 of the 110 students were surveyed, so the findings may be affected. 15-20% of female students

specialized in computer science, which was 87% male, just 8% of the student population was female. 5% of the participants completely avoided the question. Approximately 29% of students were in their third semester, while 50% were in their second semester. Students made up 9% of those in their fourth and 8% of those in their fifth semesters. In their tenth semester, only 2% of the students were present. This supports the notion that most individuals were just commencing their undergraduate studies.

Before beginning the first semester, students were asked to assess their past computer science knowledge from a wide range of possible areas. There were no students that answered no to any of these questions. Only around 50% of students had formal computer science training prior to entering college, and the majority of their previous knowledge was acquired through private sources [39]. Working in the sector did not provide any students with prior expertise. Therefore, the university provided the first formal education that the other half of the students received.

Many CS students find it difficult to develop the understanding and analysis abilities that are essential for software development, according to Edwards [40]. In order to overcome this, they advise making junior-level school curriculum exams and establishing an automated testing instrument that will give students feedback fewer defects in the students' code according to research [41] comparing courses 45% with and without TDD.

Prof. CI is a revolutionary way of programming exercises and instructing test-driven development that Matthies et al. [42] suggest. It makes use of modern, IDE-based training tools and modern, online courses designed expressly for learning how to code. The research found that the students were more motivated to develop more tests in subsequent tasks. By changing the curriculum of the introductory courses at our university, we want to educate junior-level students with improved code quality standards. This modification will be done largely by redesigning the assignments that must be completed during the semester and by using an IDE- and web-based tool to provide quality feedback.

2.8. Feedback systems using mutation testing

Feedback systems using mutation testing have become an important tool in software engineering. Software testing methods known as "mutation testing" include making minor "mutations" to existing code in order to evaluate the system's resilience. A feedback mechanism is employed to assess the efficiency of the mutation testing procedure and inform developers of the parts of the code that still require development [43].

One example of a feedback system used for mutation testing is the Mutational Divergence Score (MDS) [44]. The MDS measures the degree of divergence between two versions of a program using a combination of both static and dynamic approaches. The MDS has been found to be effective in giving developers an indication of how well their mutation tests are working in comparison with previous versions, allowing them to focus their efforts on areas where improvement can be made.

Another feedback system proposed for use with mutation testing is the Fault-Resistant Test Generation Tool (FRTG) [45]. FRTG provides feedback regarding test coverage, fault detection, and mutation scores for each test case. In addition, FRTG also provides metrics such as mutation score distribution, fault intensity, and fault propagation rate which can help developers identify areas within their code where mutations are more likely to occur

and require additional testing efforts. FRTG has been proven to be effective in providing detailed insights into how well mutation tests are performing, enabling developers to make informed decisions on where they need to focus their efforts when improving existing tests or developing new ones.

PiTest was created to offer thorough feedback on test results by adding minor "mutations" to a program's source code [46]. When using PiTest, developers specify which parts of their source code they want to mutate, as well as criteria for detecting errors in test results (e.g., timing differences). After generating mutations based on these specifications, PiTest runs individual tests (or sets of tests) against each mutated version and compares their expected output with actual output; any discrepancies are flagged as errors and reported back to developers with detailed information about where in their code those errors occurred. In addition to providing this feedback quickly during development cycles, it also helps developers identify parts of their code that need further testing before release. Several studies have used PiTest to evaluate existing software tests and provide feedback on which parts of their code require additional testing effort. For example, according to research by Fujii et al., utilizing PiTest and mutation analysis to find problems in Java applications that aren't caught by current unit tests is beneficial; it identified 20% more bugs than manual inspection alone did.

Another feedback system that has been proposed for use with mutation testing is the Adaptive Automated Reasoning System (AARS) [47]. In order to identify possible flaws in the program being tested, AARS analyses data from an existing test suite using machine learning methods. Based on its analysis, AARS then offers feedback on which areas of the code are more prone to mistakes and recommends relevant actions that should be performed to enhance the calibre of tests being developed for this specific piece of code. AARS has been found to be effective in helping developers identify areas within their code where additional attention needs to be paid when creating tests and can significantly improve their overall efficiency when writing new tests or debugging existing ones.

According to a structure of research, giving students individualized and automated feedback during programming projects may be able to help them develop their coding abilities and comprehend subjects more fully. Furthermore, existing tools such as PiTest and Unit Test Generator can be used for quickly generating test cases from mutations or unit tests from mutants, respectively, which allows programmers to generate customized feedback without having to manually create test cases or unit tests from scratch. Additionally, my proposed system for generating automated feedback based on mutation testing has been tested and found to provide greater insight into a student's code than traditional approaches such as static analysis or manual inspection. Taking all these findings into account, it is clear that utilizing customized feedback systems during programming assignments can help instructors provide more effective guidance while still saving time.

2.9. A Research on the Impact of Feedback to Programming Students

Formative feedback has been shown to be an effective method that is useful for programming student learning and vital to enhancing knowledge and skill acquisition. Offering students with constructive feedback has been shown to be effective [48] Throughout the learning

process and in the context of certain learning activities, formative feedback might flag a gap between a learner's present level of performance and some intended level of performance [49]. It is anticipated that the greatest influence on students' comprehension would result from encouraging them to reflect on their work while they are actively interacting with the topic and activity [50]. Finding a solution to the problem might also encourage higher degrees of educational endeavors [51]. Students are able to evaluate which areas of knowledge they need to investigate in greater depth and which aspects of their thinking need to be modified with the assistance of this vital information. In addition, learners, particularly novices, students who are having difficulty and students who are performing poorly, can have their cognitive load efficiently reduced by using formative feedback [52]. According to the findings of previous research [53], it appears that students are in particular need of guidance when they become stuck.

A substantial influence has been exerted on educational evaluation by the profusion of computer-related technologies that are readily available to the public. According to [54], there has been an increase in the usage of technology to facilitate the delivery or submission of assignments as well as the medium for providing feedback. For instance, the implementation of learning management systems (LMSs) or course management systems (CMSs) can alleviate the high workload associated with the submission and grading of assignments (e.g., Moodle, Blackboard, WebCT, Canvas) as well as the detection of plagiarism (e.g., Turnitin) [55], among other issues. After students submit their assignments or responses, teachers can manually analyze the student's writings or responses using these systems, evaluate how well the students did on their assignments, and provide feedback or comments to the students online.

Techniques from the fields of data mining and natural language processing (NLP) are increasingly being utilized in the field of education, particularly in the area of automatic educational assessment, as a result of the proliferation of computer-based educational technologies. Implementing computer-based feedback could be an alternative to the traditional method of receiving comments from teachers [56]. A variation of computer-based frameworks or tools has been designed to automate the process of scoring and providing feedback to cater to the requirements of a variety of different writing contexts. These systems and tools have been developed by drawing on the multidisciplinary insights found in linguistics, computer science, and educational data mining. According to what is suggested in [57], both scores and feedback in these systems are typically based on the linguistic characteristics of the student discourse. These linguistic characteristics include, but are not limited to lower-level mistakes in response (such as spelling or grammatical errors in written responses); discourse organization and structure of a piece of writing; and relevance of the discourse to the question that was asked.

Many different kinds of automatic programming feedback have been developed by researchers and programming practitioners. Compilers provide the most fundamental kind of syntactic feedback known as error messages, which have the potential to be significantly improved by adding text that is both clearer and more specific [58, 59]. In addition, the majority of environments for practicing programming, such as CodeWorkout [60] and Cloudcoder [61], provide students with the opportunity to receive feedback by running their own code through a series of test cases, each of which can either succeed or fail. Other auto graders, such as [62, 63], provide feedback in block-based languages by employing static analysis. Block-based languages do not make use of compilers.

2.10. Conclusion

This literature review has explored several models and automatic feedback mechanisms related to developing an automatic feedback system capable of generating high-quality code by utilizing metrics and mutation testing techniques. I have seen how important it is when creating such a system, to consider readability, documentation, and sufficient testing since these are crucial elements that need consideration. I have also studied how giving timely personalized hints helps students become more self-directed learners and the impact of providing incentives on student performance.

In conclusion, by exploring the various research studies related to feedback systems, automated testing tools, mutation testing tools, and other technologies used for generating test cases from mutations and unit tests from mutants. I have identified potential gaps that exist in the application and implementation of feedback systems for early learners of object-oriented programming with basic knowledge of JUnit Testing. Additionally, I have explored how mutation testing can be used to generate automated feedback hints to the students.

3

Implementation

This chapter describes the implementation process of the automated feedback system and details on how the services and components of the project were implemented. It also provides diagrams of class diagrams, sequence diagrams, etc that help reader understand the system. It also explains the language, libraries, folders, and resources used in the application. Moreover, this chapter also provides an overview of how the feedback system is structured and how it works.

In the first section, the project structure is briefly explained, and graphical representations are provided. In the next step, the project components are analyzed and demonstrated using diagrams, followed by system flow diagrams.

3.1. Project Structure

The project structure for the feedback system is organized in order to make it easier to understand. Java packages are a way of organizing and managing related classes, interfaces, subclasses, exceptions, errors, and enums. This organized project structure makes it easy to locate the various components necessary for generating feedback on programming assignments with mutation testing. The components of the feedback system are divided into packages that are further subdivided and located in the `sre/main/java`. Additionally, there is also a test package that contains unit tests for each of the main packages in order to ensure that all functionality works as expected.

Java provides some built-in packages which we can use but we can also create our own (user-defined) packages by defining a directory with the same name as the package and placing classes inside it. The advantages of using Java Packages are that they make easy searching or locating of classes and interfaces; avoid naming conflicts; implement data encapsulation (or data-hiding); provide controlled access; allow for the reuse of classes contained in other packages, and uniquely compare classes in other packages. There are two types of packages in Java: Java API packages or built-in packages, which provide a large number of classes grouped into different packages based on a particular functionality, and user-defined packages which are created by the user.

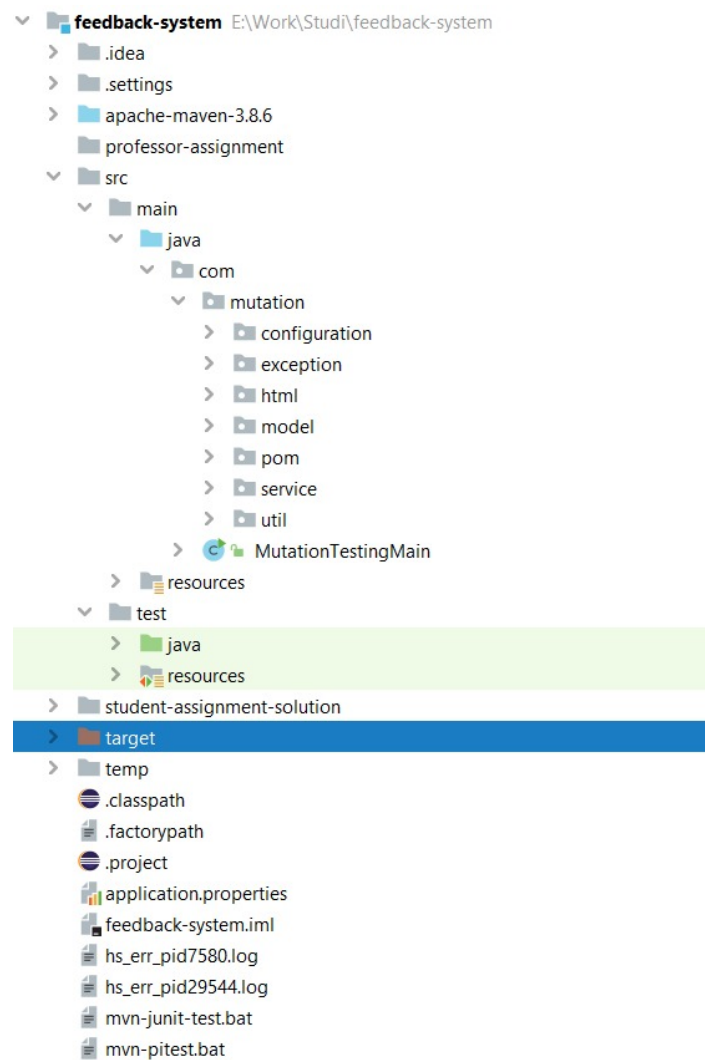


Figure 3.1: Project structure

3.1.1. Technologies and Tools used

Technologies that are used in this project include:

intellij/eclipse sts, Java version 17, Maven, JUnit, PiTest, JSoup HTML Parser, Commons-IO, Zip4j, EclEmma, Log4j and Dropbox.

All the dependencies can be found in the “pom.xml” file.

A Project Object Model (POM) is the core unit of Maven’s build system. It is an XML document that contains information about the project, such as its configuration details, and provides default values for most projects. POMs are used by Maven to construct a project’s overall build process, allowing developers to easily manage dependencies, build processes, and other aspects of the project. For the vast majority of projects, it comes with default settings.

- Java 17: As the newest long-term support (LTS) release for Java under its six-monthly release cycle, Java 17 is the latest release under that cycle. It was developed in

collaboration with Oracle engineers as well as the entire Java community via the OpenJDK Community and the Java Community Process (JCP).

- JUnit: As the name implies, JUnit is a framework for unit testing for the Java programming language. The JUnit framework is one of a family of unit testing frameworks that are collectively known as xUnit that originated with SUnit, and has played a very important role in the development of test-driven development. When JUnit is compiled, it is linked as a JAR file and it is included in the build process.
- PITEST: PITEST is a state-of-the-art mutation testing system that provides Java and the JVM with standard coverage for any changes made during development. This tool has a fast, scalable configuration, and it integrates well with modern tools for testing and building.
- Mutation testing: The purpose of mutation testing is to improve the adequacy of tests and to uncover defects in a program by testing it against a range of different mutations. This is based on the idea that the production code is going to be changed dynamically and, as a result, the tests will fail.
- Jsoup: This open-source Java library is designed to parse, extract, and manipulate data stored in HTML files.
- Commons-IO: This is a collection of utilities for developing IO functionality as a part of Apache Commons IO. Among the six main areas in this package, there are six main groups: io - This package defines utility classes for working with streams, readers, writers, and files. File Comparator - This package provides different implementations of File Comparators for use with various file types.
- Zip4j: This is the most complete Java library for zip files or streams. It also supports zip encryption.
- Maven project manager tool. The Apache Group created the well-known open-source build tool Maven to create, publish, and distribute various projects simultaneously for improved project management. It is a tool that allows programmers to create and document the dynamic framework.
- Log4j: It is a component of the Apache Software Foundation's Apache Logging Services project written by Ceki Gülcü.

3.1.2. Architecture

Dropbox

The feedback system uses Dropbox to store project files that have been submitted by students. Dropbox has four main folders. They include Submitted Project, the Project Success, projects processing and the Project Failure.

- Submitted Project: This folder contains projects that have been submitted by students.
- Project Success: This folder contains project-passed Junit test and PiTest. Rclone moves these projects from the Local Server to the Project Failure folder.
- Project Failure: This folder contains projects that have not passed the Junit test and PiTest. Rclone moves these projects from the Local Server to the Project Failure folder.
- projects processing: this folder contains projects for processing.

Local Server

This is a server where Feedback-Application is deployed and Rclone Program is installed. The Server has three parts that include: Folder directories, Rclone, and Feedback-App.

Folder Directories

The Local Server has four folder directories, which include Submitted Project, Project Processing, Project Success, and Project Failure.

- Submitted Project: This folder contains projects submitted by students. The projects are moved from Dropbox to the Local Server by the Rclone program. The Feedback application reads this folder to find the submitted project that needs to run the test.
- Project Processing: After Feedback application has found the projects in the Submitted Project folder, the Feedback application then moves those projects to the Project Processing folder to perform the test.
- Project Success: If that project passes both the Junit and PiTest test, the project is moved to the Project Success folder.
- Project Failure: If that project fails to pass both Junit test and PiTest, it will then moved to the Project to the failure folder.

Rclone

Rclone is a command line program to manage files on cloud storage. It is a feature-rich alternative to cloud vendors' web storage interfaces and supports over 40 cloud storage products including S3 object stores, business & consumer file storage services, as well as standard transfer protocols. Rclone's familiar syntax includes shell pipeline support, and dry run protection. It is used at the command line, in scripts, or via its API. Rclone has been used in this feedback system to move files between the Local Server and Dropbox, ensuring anonymity. Rclone is a command line program that is used to automate the process of moving files between these two locations.

Rclone does the heavy lifting of communicating with cloud storage for users to move files between Dropbox and Local Server in such a way that it maintains file integrity by checking MD5 SHA1 hashes at all times as well as preserving timestamps of file. Scheduled events help trigger Rclone program which moves files from Dropbox: submitted project to local: submitted project; local: project success to Dropbox: project success; local: "project failure" to

Dropbox: "project failure". This helps ensure that projects are moved efficiently between the two locations without any data loss.

The system flow begins with a Scheduled timer event which triggers the Rclone program to execute its steps. The first step is for Rclone to move files from Dropbox: submitted project to local: submitted project. This moves the files from Dropbox into the Submitted Project folder on the Local Server.

The second step is for Rclone to move files from local: project success to Dropbox: project success. This moves all projects that have passed both Junit test and PiTest from the Local Server into the Project Success folder on Dropbox.

The third and final step is for Rclone to move file from local: "project failure" to Dropbox: "project failure". This moves all projects that have failed both Junit test and PiTest from the Local Server into the Project Failure folder on Dropbox. After this, the system flow will end.

Rclone is mature, open-source software originally inspired by rsync and written in Go. It mounts any local, cloud or virtual filesystem as a disk on Windows, macOS, Linux and FreeBSD and also serves these over SFTP, HTTP, WebDAV, FTP and DLNA. Third-party developers create innovative backup, restore, GUI and business process solutions using the rclone command line or API.

Feedback Application

The Feedback application is designed to help students gain a better understanding of their coding assignments. It reads the submitted projects and performs Junit test and PiTest. If the project passes both tests, it is moved to the Project Success directory. However, if it fails either of the tests, it is moved to the Project failure directory.

The Feedback application then generates a report that provides hints and feedback to the students regarding how they can improve their code in order to pass the PiTest. This report helps students understand what areas they need to focus on in order to be successful with their projects.

The Feedback application also scans submitted projects for any folders; if found, these are moved to the Project Processing directory. From here, each project folder is tested one by one until all have been tested and processed.

The Feedback application is an invaluable tool for students who may be struggling with coding projects or who just want to further hone their skills. It allows them to quickly identify issues with their code and take actionable steps toward improving it, resulting in better outcomes for their projects.

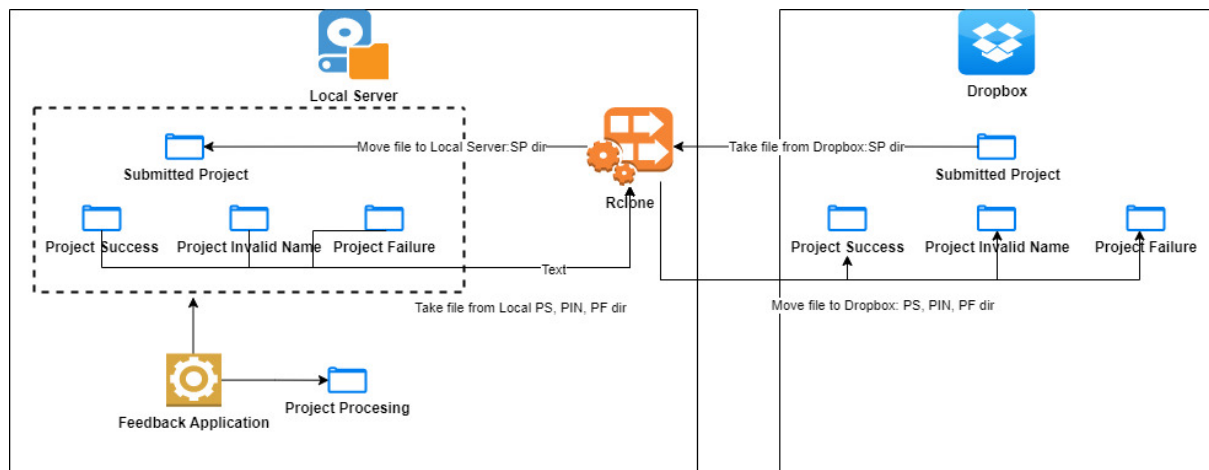


Figure 3.2: Architecture

3.2. Service Flow

3.2.1. Main flow

This is the flow that shows the steps and stages that take place when a student solution is submitted to the feedback system. These steps are explained below:

- **Scheduled:** This is the section that shows a timer event start. The user in the configuration file will schedule it. Example: Run every 8 am from Mon-Fri. Whenever the timer reaches the main flow it will be triggered to start.
- **Check local: Submitted project directory.** It is a stage that checks whether the submitted projects directory has a new folder or not.
- **Is have new file:** This is a condition activity. It checks whether a new project has been submitted, and move to the next stage. If no new project has been submitted the system will end the process.
- **Move file to local: Project processing directory.** At this stage, the project folder is moved from the submitted project to the project processing directory.
- **Execute Junit test:** Here, the Junit tests from the project folder are performed one by one in the project processing directory.
- **Junit test pass:** Here if the project file passes the Junit test, they proceed to the Execute PiTest stage. However, if the Junit test fail, they move to the Move file to local: project failure.
- **Move file to local: Project failure.** This is where project files move when they fail to pass Junit test move.
- **Execute PiTest:** This is where PiTest is performed on project files that passed Junit test.
- **PiTest pass:** Here if the project file passes the PiTest, they proceed to the Move file to local: project passed. However, if the Junit test fail, they move to the Send email notification for the report test stage.

-
- Move file to local: project success. This is the stage where the project file that passes PiTest moves the project folder from the project progressing directory to the project success directory.
 - Send email notification: This is the stage where a report is generated, then sent to the Professor and the Student who owns the project. After this, the process ends.

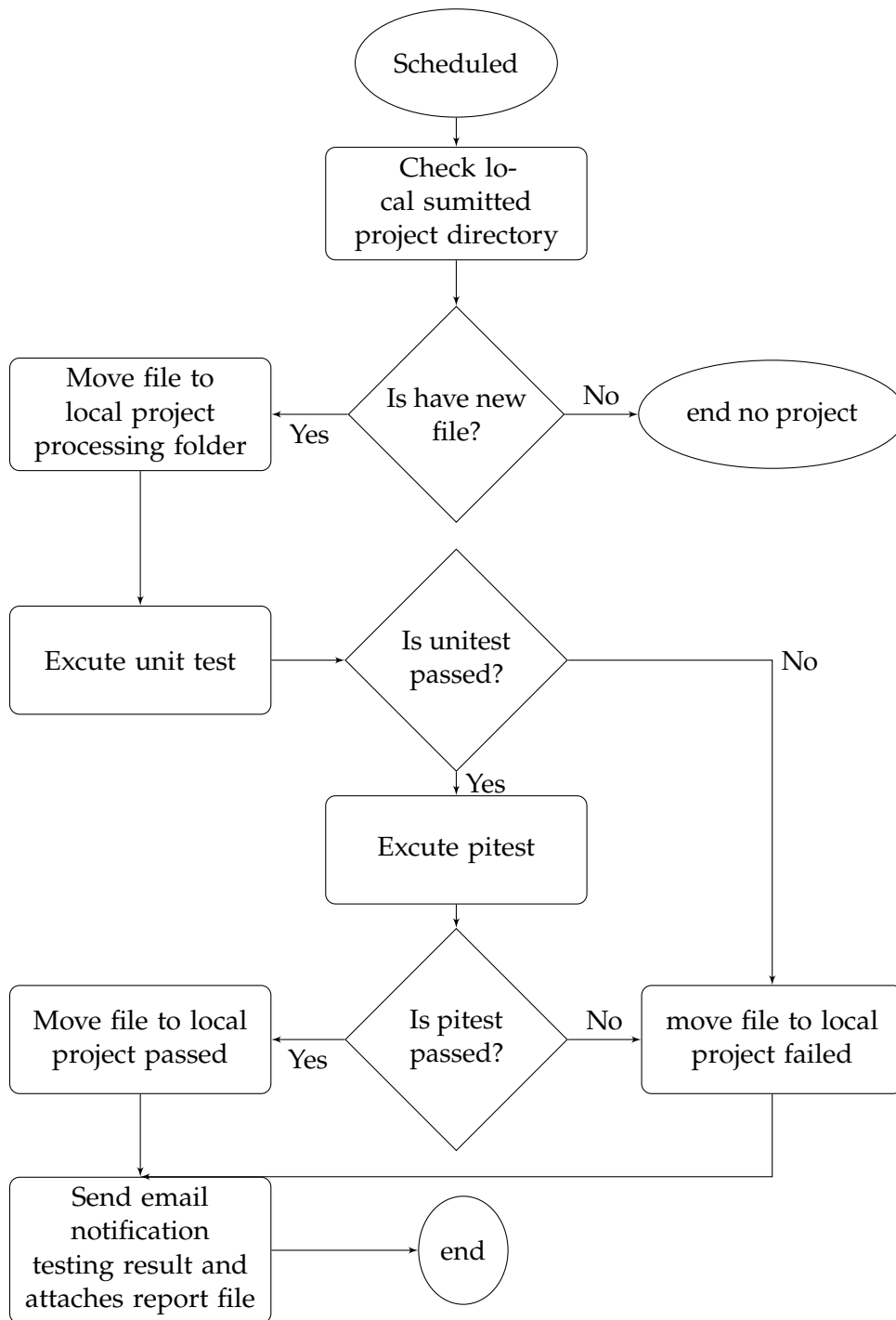


Figure 3.3: Main Flow

3.2.2. Junit test Flow

The Junit flow starts with the `JobStartMainFlow` function, which calls `setupSelectedProjectTesting()` method on `MutationTestingService` class to set up and perform unit tests and mutation tests on the project. The `createCommandForUnitTestingWithFullProjectPath()` method is then called on `MutationTestingService` class to create commands for unit testing. Then, `writeCommandToFile()` method is called to write the command to a file called “mvn-junit-test.bat”. After that, `performJUnitTestingProcessBuilder()` method is called to execute the unit test. If the unit test fails, it will return false, otherwise it will move to the next step of executing mutation testing. This is a flow that is triggered by the main project flow and it involves five main steps that are summarized below:

- **Create command for Junit test:** This is the stage where the command is created to run the Junit test. The command is then saved to a file type “.bat”, which is later executed by the system.
- **Perform Junit test:** This is the stage where the file generated from step 1 is executed.
- **Junit test passed:** Here, Junit test is performed on the files, and if they pass, they are moved to the PiTest Performing stage. However, if they fail, they are moved to the Return status failed at Junit stage.
- **Return status failed at Junit test:** This is where files that have failed the JUnit test are moved to the failure directory and the Junit test flow ends.
- **Perform PiTest flow:** This is where PiTest is performed on the files that passed Junit test.

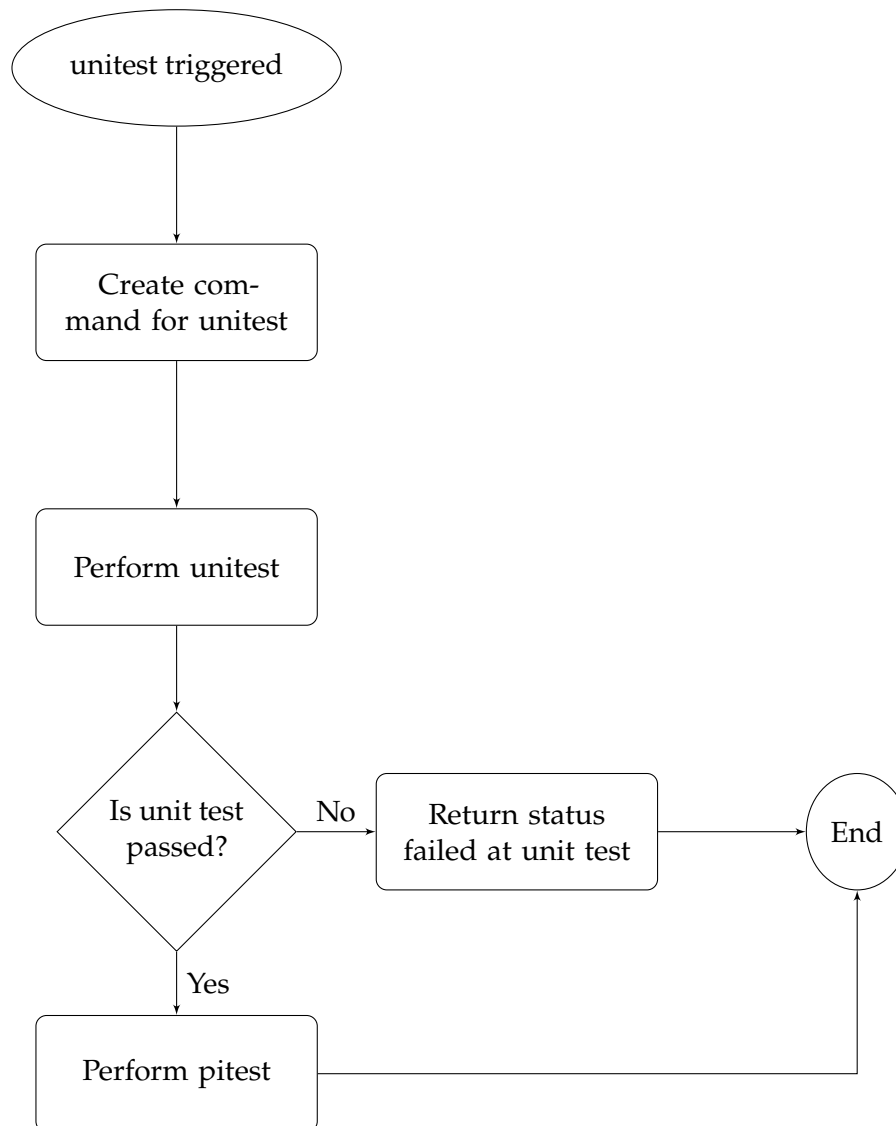


Figure 3.4: Junit Test Flow

3.2.3. PiTest flow

. The `includePiTestDependencyInPomFileFullProjectPath()` method is first called on `MutationTestingService` class to include PiTest dependency in pom.xml file. Then, `writeModifiedModelToFileWithFullProjectPath()` method is used to write PiTest dependency to pom.xml file in `ModifyPomXml` class. In `ModifyPomXml` class, the `populateModifiedModelWithFullProjectPath()` method is used to parse pom.xml file to a `Model` object. The `addPiTestDependency()` method and `addPlugin()` method are used to add PiTest dependency alongside with the plugin respectively in pom.xml file before returning it back to `MutationTestingService` class. Once it is returned, the `createCommandsForMutationTestingWithFullProjectPath()` method is called to create commands for mutation testing and the `performMutationTestingUsingProcessBuilder()` method is used to execute the mutation test before returning true if it is

successful or false if unsuccessful. By following this process, students can receive feedback about their assignments in a timely manner regarding their code quality based on the results of their test suite.

The PiTest flow has eight main function. The flow is triggered by the main process flow.

- Scan pom.xml file: This is the first stage of the PiTest flow where the project folder is scanned to read the content of “pom.xml” file.
- Is PiTest dependency Injected: Here, the “pom.xml” file is checked to see if it has PiTest dependency. If PiTest dependency is present it will proceed to Create the command for PiTest stage. However, if the PiTest dependency is not present it moves to the Inject PiTest dependency stage.
- Inject PiTest dependency: Here, the PiTest dependency is added to the “pom.xml” file.
- Create Command for PiTest: This is the stage where the command is created to run the PiTest. The file is saved to a file type “.bat”.
- Perform PiTest. This is where the PiTest is performed on the test suite generated from the Create Command for PiTest.
- Is PiTest pass: Here, if the test suite pass the PiTest stage, they proceed to the Return status success at PiTest. However, if the test suite fail PiTest, they are moved to the Return status failed at PiTest.
- Return status failed at PiTest: Here, the status of the test suite that failed to pass PiTest is displayed and then the process ends.
- Return status success at PiTest: Here, the status of the test suite that passed PiTest will be displayed and then the process ends.

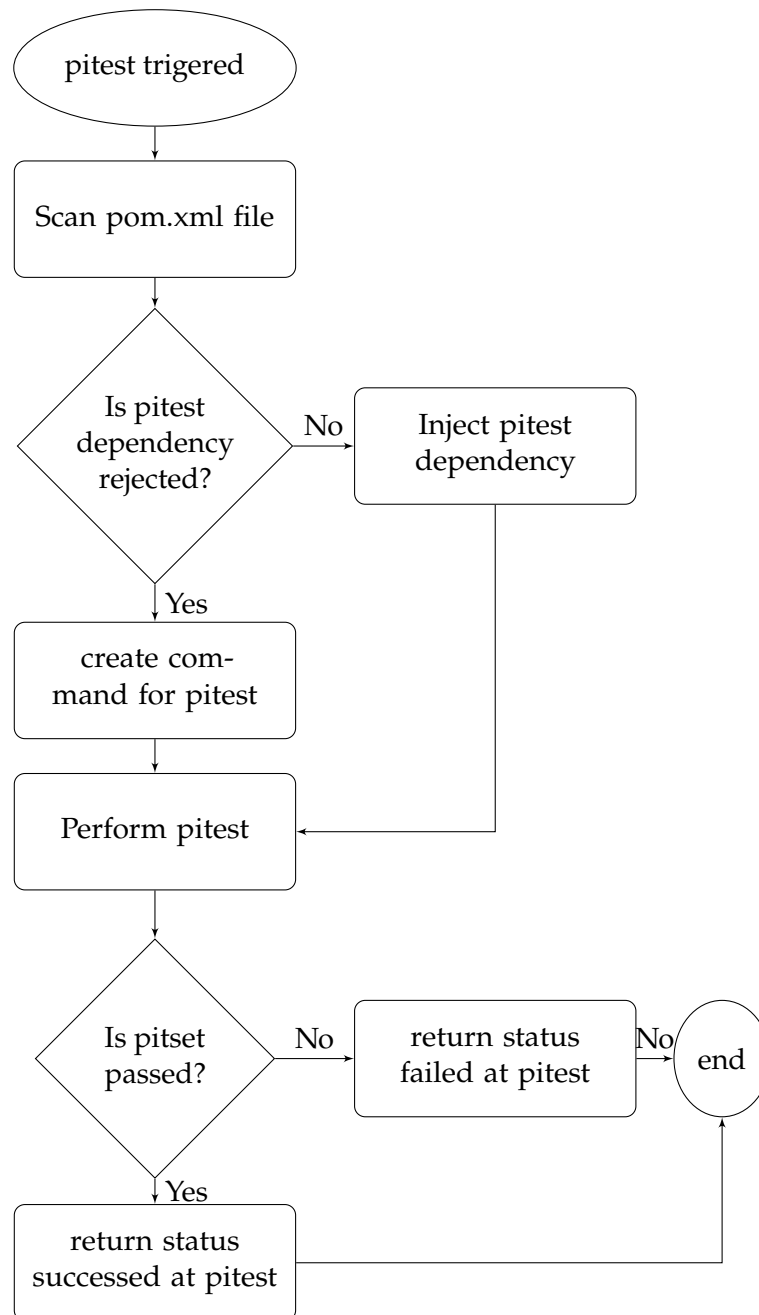


Figure 3.5: PiTest Flow

3.2.4. Moving File Flow

- Scheduled: This timer event starter is configured to trigger the Rclone program.
- Rclone move files from Dropbox: Move submitted project to local submitted project directory. Rclone moves the file submitted project directory from Dropbox to the Local Server in the submitted project directory.
- Rclone move files from local: Move project success to Dropbox project success directory.

Rclone moves the file project success directory from Dropbox to the Local Server in the project success directory.

- Rclone move files from local: Move project failure to Dropbox project failure directory. Rclone moves the file project failure directory from Dropbox to the Local Server in the project failure directory. After this the system flow will end

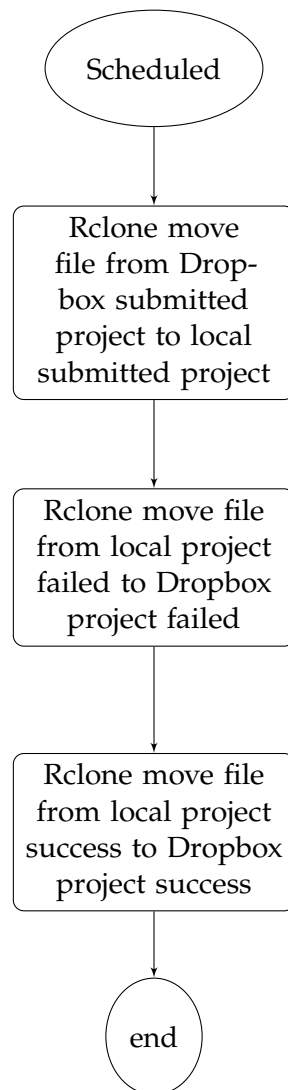


Figure 3.6: Moving Flow

3.3. Project Diagram

This use case diagram below give a sense of orientation of the feedback system. It provides detailed insight into the structure of the system. At the same time it offers a quick overview of the functions happening in the system as well as the two actors involved.

3.3.1. Use case diagram

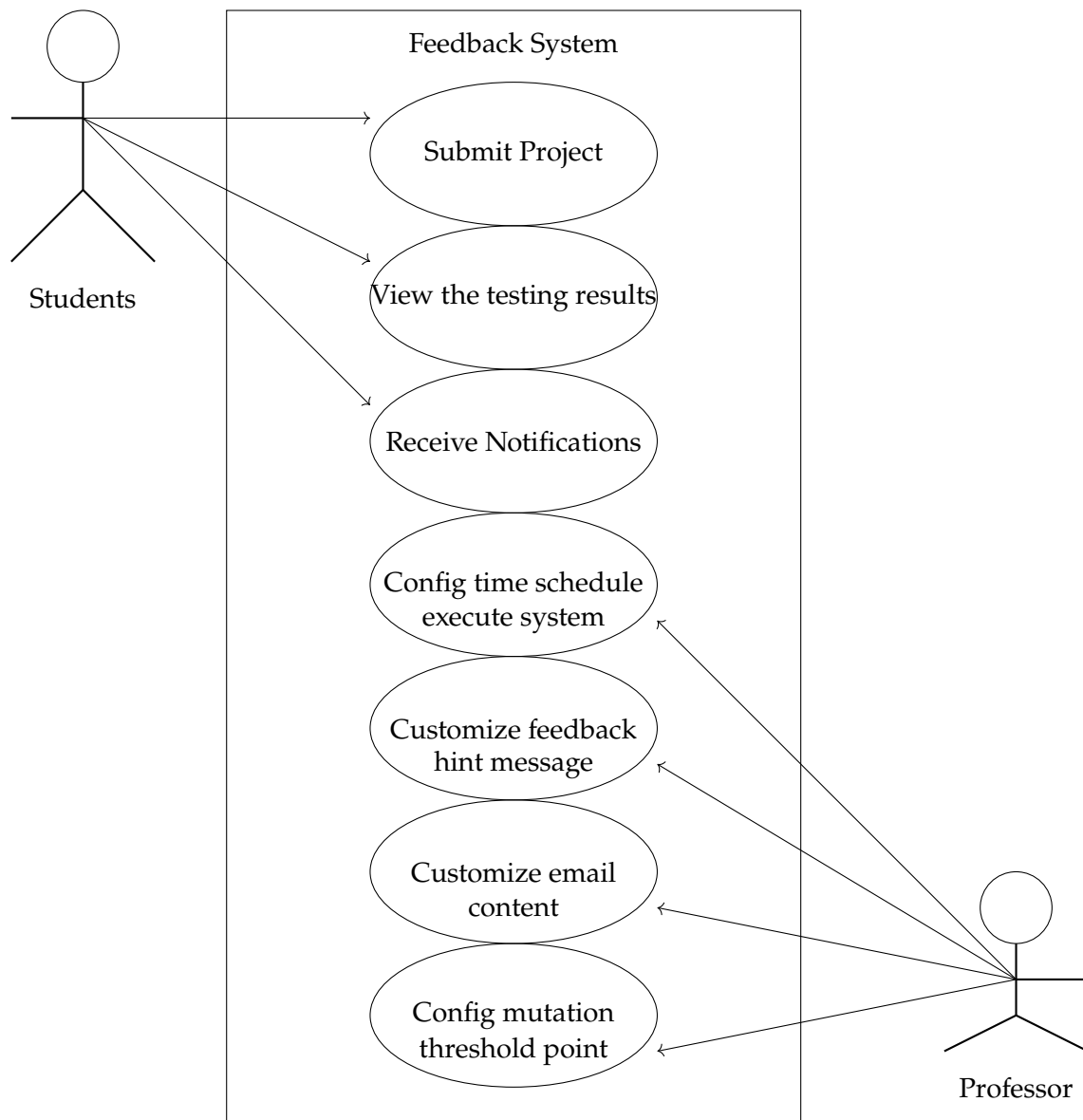


Figure 3.7: Use Case Diagram

Mutation Testing Main class

The Main class of this application will be executed first when the application executes.

Properties Configuration class

This class is used for initial configuration for the whole project. The static factory constructor from application Properties file() loads configuration from the "application.properties" file, then sets it to an instance of the Properties Configuration object.



Figure 3.8: Properties Configuration class

Job Start MainFlow class

This class is implemented from Job class, which belongs to Quartz dependency. JobStartMainFlow class is a presentation of the main flow diagram.



Figure 3.9: Job Start Main Flow class

Dependencies:

- ProjectDirServiceImpl.class
- Email ServiceImpl.class
- Mutation Testing Service.class
- Html Reader.class

3.3.2. Project DirServiceImpl class

This class is implemented from ProjectDirService class which provides service related to ProjectDir(submitted project dir, processing project dir, failure project dir and success project dir).

ProjectDirServiceImpl		
m	ProjectDirServiceImpl()	
m	moveProjectFromSubmittedToFailureDir(String)	void
m	getProjectSuccessPath(String)	String
m	getProjectSubmittedPath(String)	String
m	isProjectNameValid(String)	boolean
m	moveProject(String, String)	void
m	moveProjectToSuccessDir(String)	void
m	moveProjectToProcessingDir(String)	void
m	moveProjectToFailureDir(String)	void
m	getProjectFailurePath(String)	String
m	getProjectProcessingPath(String)	String
p	listProject	String[]
p	haveNewProjectSubmitted	boolean

Figure 3.10: Project DirServiceImpl class

Dependencies:

- Properties Configuration.class
- Project DirService.class

Email ServiceImpl class

This class is implemented from Email Service class. This class provides service related to email actions such as send Email(), send Email With Attachments().

EmailServiceImpl		
m	EmailServiceImpl()	
m	sendAttachmentEmail(String, String, String[])	void
m	sendEmail(String, String)	void

Figure 3.11: Email ServiceImpl class

Dependencies:

- Email Service.class

Mutation Testing Service.class

This class is used for setting up and performing Junit test as well as PiTest.

By setting up I mean verifying the pom.xml file and injecting PiTest dependency in cases that it is not injected.

MutationTestingService		
m	MutationTestingService(String)	
m	performJUnitTestingUsingProcessBuilder()	boolean
m	performMutationTestingUsingProcessBuilder()	void
m	writeCommandToFile(List<String>, String)	void
m	createCommandsForUnitTestingWithFullProjectPath(String)	void
m	createCommandsForMutationTesting(String)	void
m	setupSelectedProjectAndPerformTesting()	boolean
m	createCommandsForUnitTesting(String)	void
m	createCommandsForMutationTestingWithFullProjectPath(String)	void
m	includePiTestDependencyInPomFileFullProjectPath(String)	void
m	copyInputProjectToTempFolder(String)	void

Figure 3.12: Mutation TestingService class

Dependencies:

- Modify PomXml.class
- ProjectDirServiceImpl.class

HtmlReader class

The HtmlReader class is used to read the .html files generated by the PiTest mutation testing tool, extract the results, and write them to a file called "feedBackReport.txt" which can then be sent in an email.

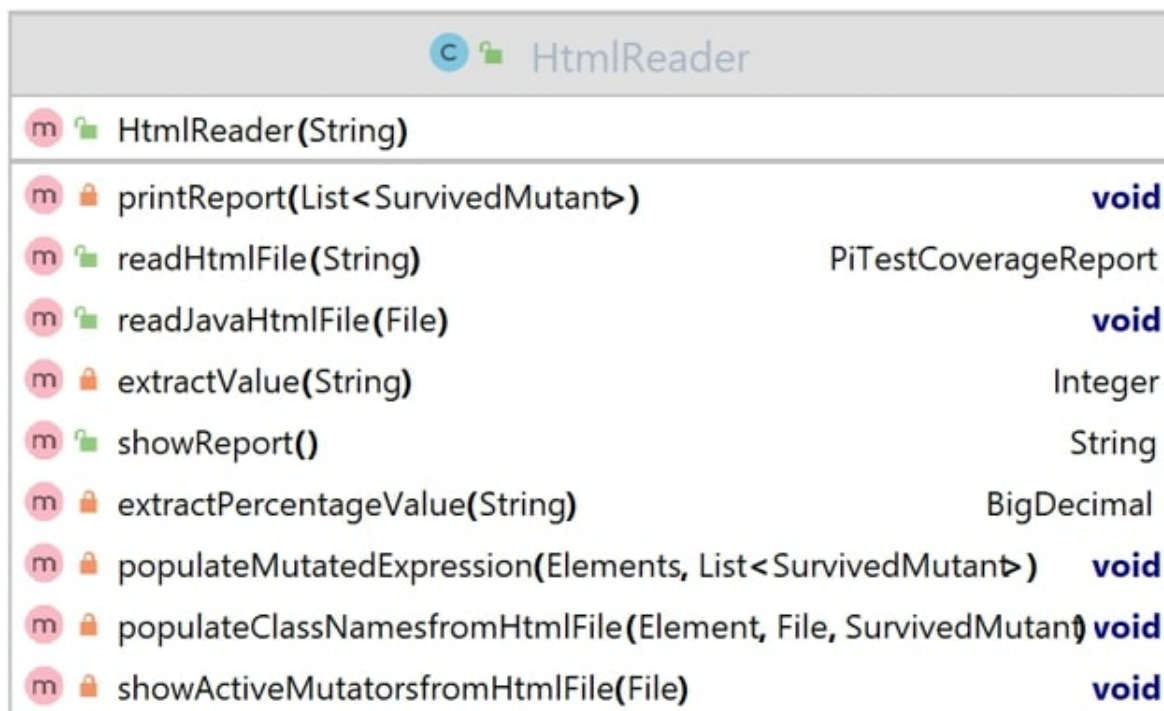


Figure 3.13: HtmlReader class

Dependencies:

- Mutation Sample Message Service.class
- ProjectDirServiceImpl.class
- PiTest Coverage Report.class
- Feed back Message Service.class

ProjectDirService class

This is an interface that defines the methods for the ProjectDir object.

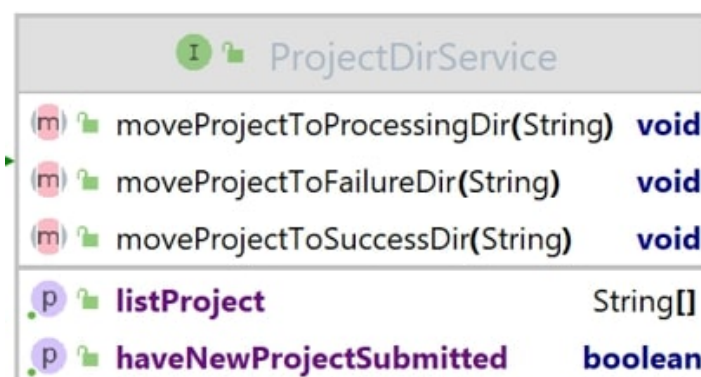


Figure 3.14: Project DirService Interface

Email Service class

This is an interface that defines the methods for the Email object.

Modify PomXml.class

This class is used to read the pom.xml file in submitted project, add PiTest dependency and PiTest plugin to that pom.xml file.

Dependencies:

- InvalidProjectExeption.class
- ProjectDirServiceImpl.class

ModifyPomXml		
m	ModifyPomXml(String, String)	
m	addPlugin(Model)	void
m	populateModelWithFullProjectPath()	Model
m	populateModifiedModelWithFullProjectPath()	Model
m	populateModel()	Model
m	writeModifiedModelToFileWithFullProjectPath()	void
m	writeModifiedModelToFile()	void
m	populateModifiedModel()	Model
m	addPiTestDependency(Model)	void

Figure 3.15: Modify PomXml class

Mutation Sample Message Service class

This class loads the samples of mutation and the solution for each mutation operator. These files contain samples located in the "src/main/resources" directory.

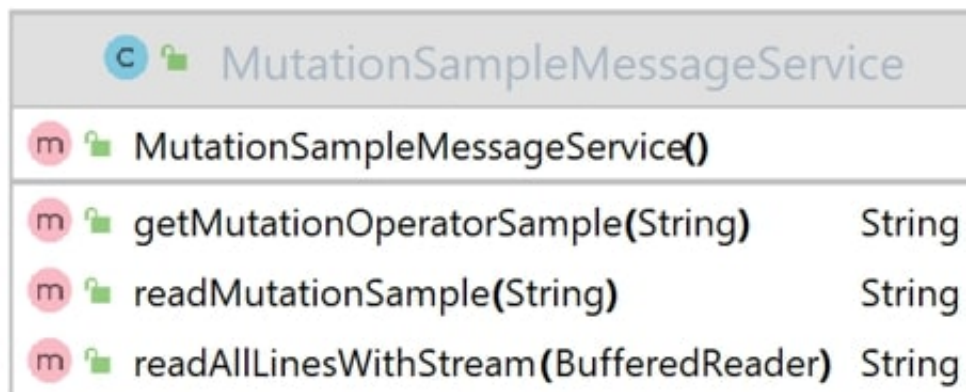


Figure 3.16: Mutation Sample Message Service class

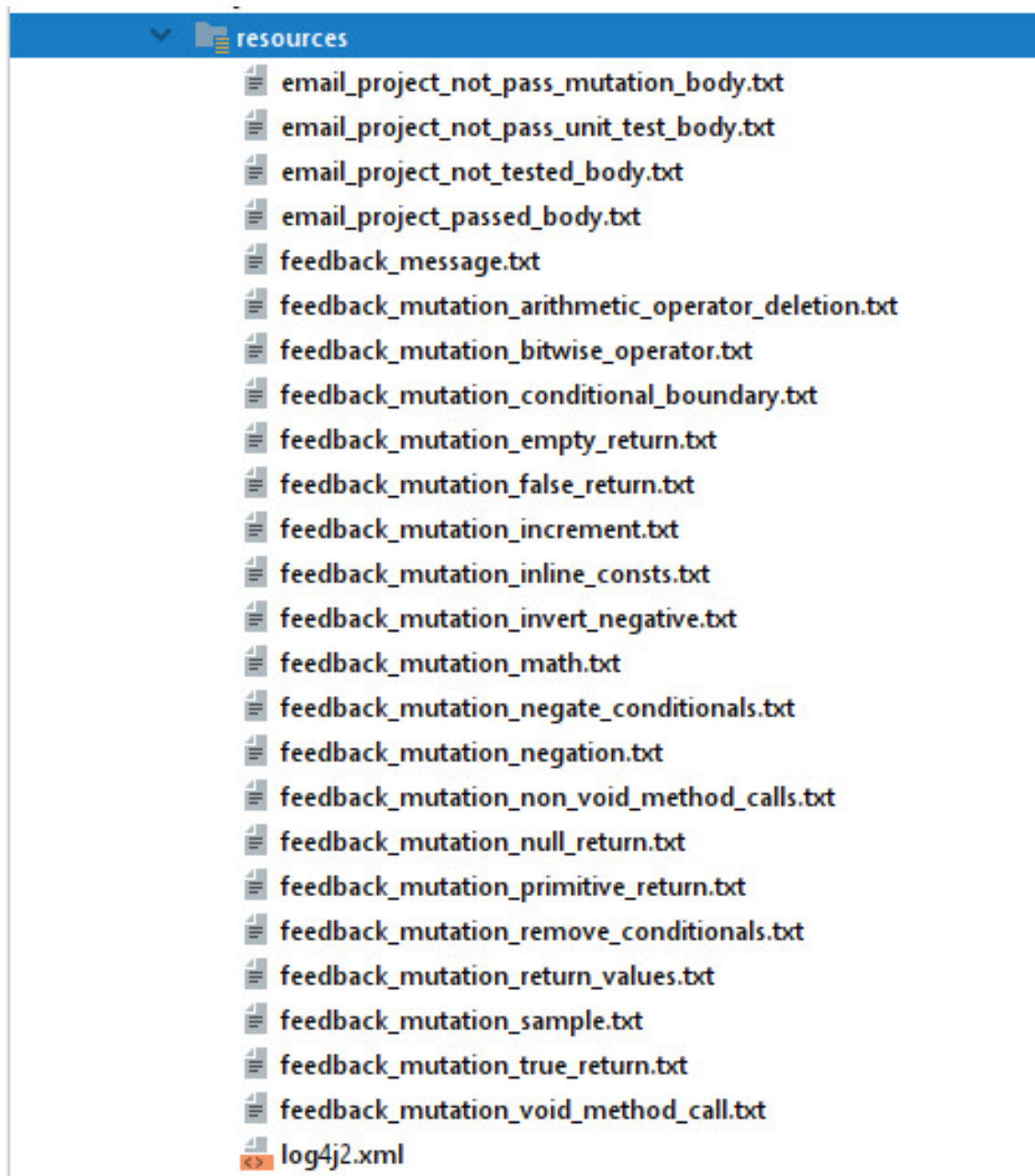


Figure 3.17: Mutation Samples File

PiTest Coverage Report class

This is the model class that presents PiTest coverage report object.

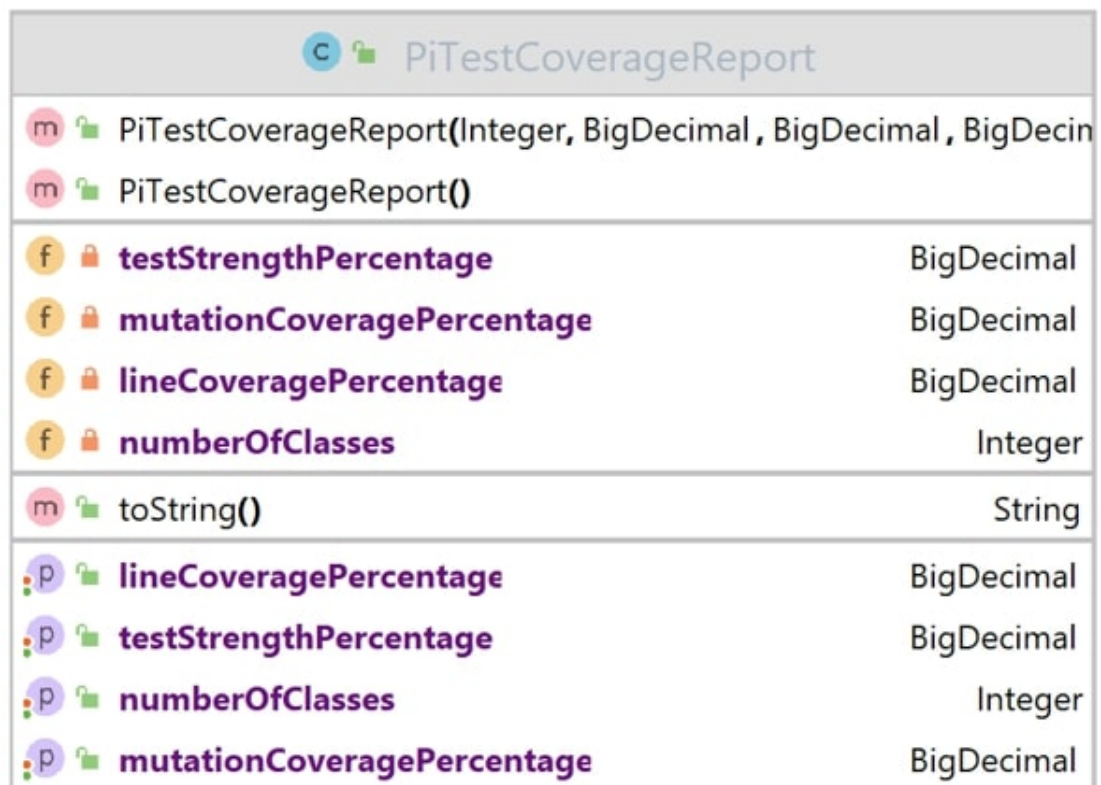


Figure 3.18: PiTest Coverage Report class model

Feedback Message Service class

This class reads the file "feedback message.txt" from the "src/main/resources" directory then generates a feedback message base on the mutation result.

FeedbackMessageService		
m	FeedbackMessageService()	
f	headerMessage	String
f	footerMessage	String
m	replaceLineNumber(SurvivedMutant)	void
m	getMutationMessage(SurvivedMutant)	String
m	replaceLineJavaCode(SurvivedMutant)	void
m	replaceOriginalExpression(SurvivedMutant)	void
m	replaceMutatedExpression(SurvivedMutant)	void
m	replacePackageName(SurvivedMutant)	void
m	generatedMutation(SurvivedMutant)	void
m	readFeedBackTemplate()	void
m	replaceMutationOperator(SurvivedMutant)	void
m	readAllLinesWithStream(BufferedReader)	String
m	replaceClassName(SurvivedMutant)	void
p	headerMessage	String
p	footerMessage	String

Figure 3.19: Feedback Message Service class

Invalid Project Exeption class

This is a custom exception handler model.



Figure 3.20: Invalid Project Exception class

Survived Model class

This is the model present for Survived Mutation object when executing PiTest



Figure 3.21: Survived Model

FeedBack Template Parameter class

This class declares configuration parameter mapping for the feedback file in “src/main/resources/feedback message.txt”

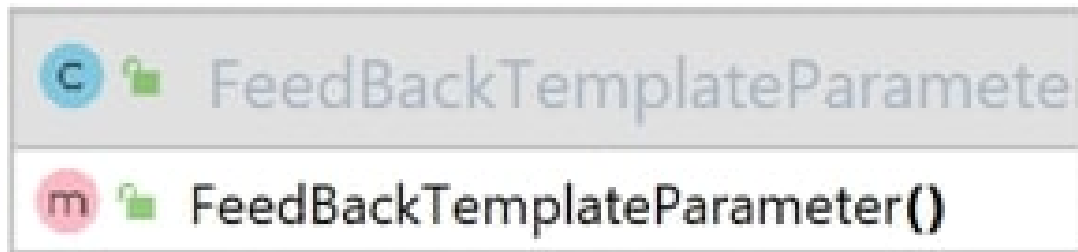


Figure 3.22: Feedback Template Parameter

Mutation SamplesParamter.class

This class declares configuration parameter mapping for mutation samples file in “src/main/resources/feedback mutation operatorName.txt”



Figure 3.23: Mutation Samples Parameter

3.3.3. Sequence diagram**Main Sequence Diagram**

The flow diagram that illustrates the steps and stages that must be followed in order for a solution to be submitted to the feedback system.

The above flow diagram illustrates the main functionalities of the Mutation Testing System. The Main class of this application will be executed first when the application executes. It starts with the Main function which trigger a scheduler, as well as loading cron expressions from an application properties file. The scheduler then invokes the Job Start MainFlow function.

The Job Start MainFlow function first calls IsHaveNewProjectSubmitted() method on the Project DirServiceImpl class to check whether there are any new projects in the ‘submitted project’ folder. If there are no new projects, it will console log a description and wait for the next scheduled time. Otherwise, it will call getListProject() method on Project DirServiceImpl class to get an array of project names.

It will then loop through these project names and check if each project name is invalid or not. If it is found to be invalid, it will call move Project From Submitted ToFailureDir()

method on Project DirServiceImpl class to move this project from the submitted directory to a failure directory. Otherwise, it will call setup Selected Project And Perform Testing() on Mutation Testing Service to set up and perform unit tests and PiTest on the project before receiving a response if the test has been completed successfully or not from Mutation Testing Service class.

If it has been completed successfully, Job Start MainFlow will call show Report() method on HtmlReader class to read the mutation testing results before sending an email notification about the results of these tests via send Email() method on EmailServiceImpl class.

Finally, depending on whether or not the mutation testing is passed, JobStartMainFlow method will either call move Project To SuccessDir() method or move Project ToFailureDir() method on Project DirServiceImpl class before looping through any remaining projects in its array until all have been tested.

Junit test Sequence Diagram

In the main flow diagram, mutation testing service sets up and performs unit tests and PiTest on the project before receiving a response if the test has been completed successfully or not

There are five steps that are followed in order to complete the process of Junit test which are triggered by the main flow that is shown in the flow diagram below.

The above Junit test and PiTest flow starts with the Job StartMainFlow function, which calls setupSelected Project Testing() method on Mutation Testing Service class to set up and perform unit tests and mutation tests on the project. This process begins by calling create Commands For Unit Testing With full ProjectPath() method on Mutation Testing Service class to create commands for unit testing. Then, the write CommandToFile() method is called to write the command to a file. After that, perform JUnitTesting Process Builder() method is called to execute the unit test. If the unit test fails, it will return false, otherwise it will move to the next step of executing mutation test.

PiTest Sequence Diagram

This diagram shows where you can find the auto generated PiTest feedback report.

The MutationTestingService provides a way for students to obtain feedback on their code quality in a timely manner. This is achieved by first including the PiTest dependency in the pom.xml file of the project. In order to do this, the ModifiedModelToFileWith-FullProjectPath() method is used to write the PiTest dependency into the pom.xml file in ModifyPomXml.PopulateModifiedModelWithFullProjectPath() method then parses the pom.xml file into a model object and adds both the PiTestDependency() and Plugin(). These are then returned back to MutationTestingService class for further processing.

Once returned, createCommandsForMutationTestingWithFullProjectPath() method is called to create commands for mutation testing and perform mutation testing using Process-Builder() method to execute the mutation test before returning true if successful and false if unsuccessful. Through this process, students can receive feedback on their projects based on the results of these tests and make necessary changes accordingly in order to improve their code quality.

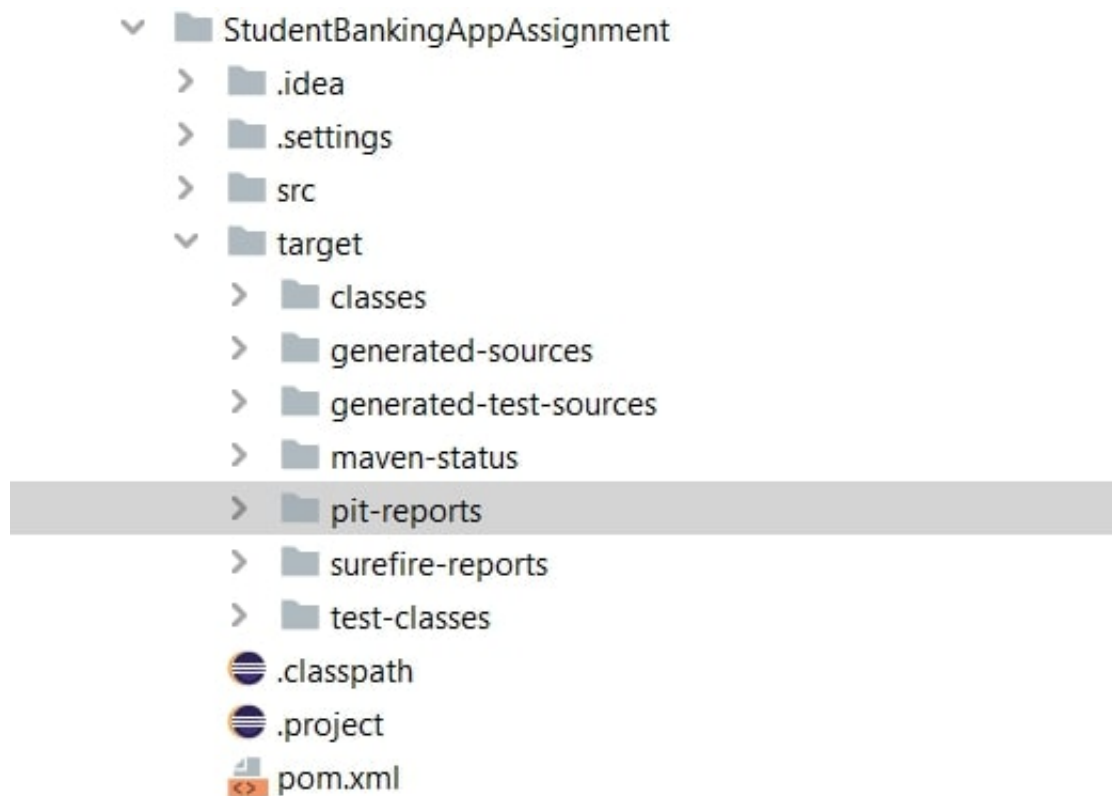


Figure 3.24: PiTest report

Extract report flow

[As discussed in the MAIN FLOW DIAGRAM, if the response has been completed successfully, Job Start Main Flow will call show Report() on Html Reader to read the mutation testing results.]

The extract report flow begins with the JobStartMainFlow function, which calls ShowReport() method on HtmlReader to read the mutation testing results. After reading the HTML files, HtmlReader extracts the report and stores it in the PiTestCoverageReport model. It also reads and prints the report details to a file named “feedbackReport.txt”.

The response of the testing is then sent back to JobStartMainFlow. Subsequently, JobStartMainFlow triggers an email notification about the results of these tests through SendAttachmentEmail() method, attaching the “feedbackReport.txt” file to EmailServiceImpl class.

Depending on whether or not the mutation testing passed, JobStartMainFlow will either call moveProjectToFailureDir() method on EmailServiceImpl class or moveProjectToSuccessDir() method on ProjectDirServiceImpl class before looping through any remaining projects in its array until all of them have been tested. This ensures that students receive accurate feedback about their projects in a timely manner.

Unit test cases

JUnit library is used to create unit test cases. Three test classes were added to cover unit testing of all logic parts in the application. These three classes are available in the package com.mutation.service under the folder src/test/java.

3.3.4. Maven library

Maven library added to the application is to execute JUnit testing and PiTesting of the submitted project.

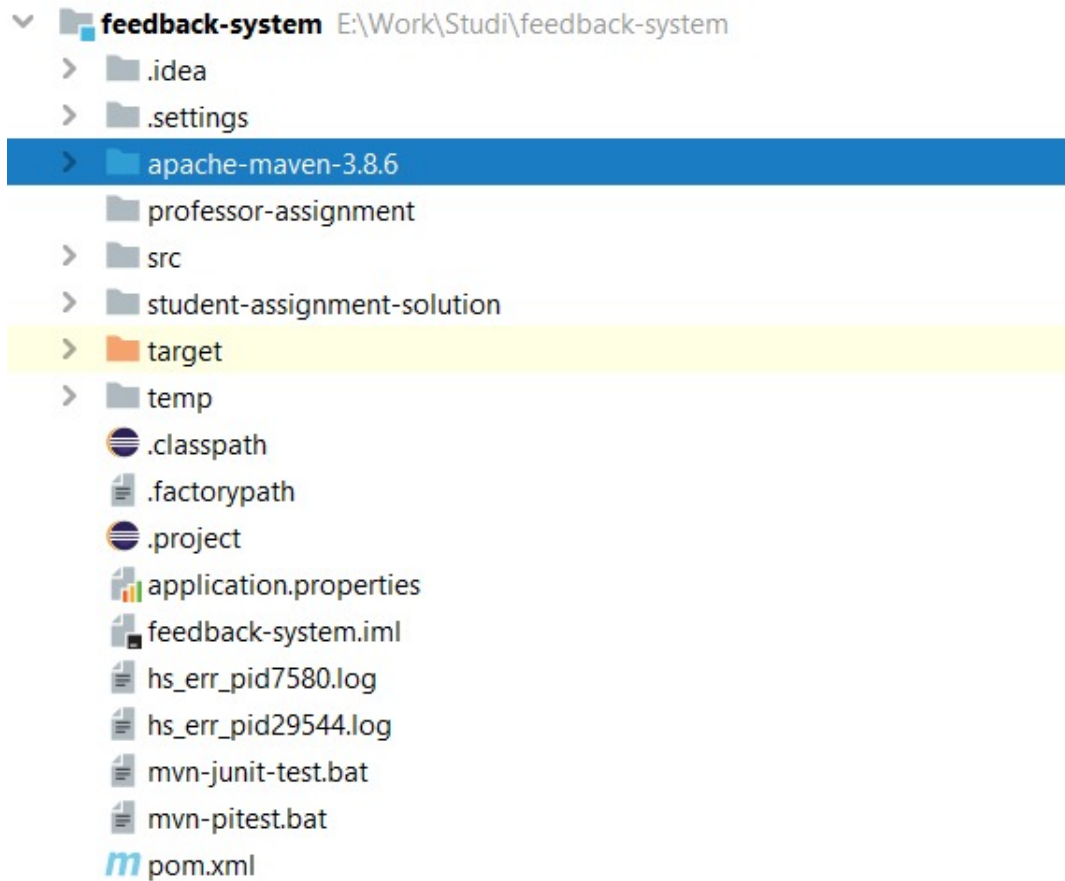


Figure 3.25: maven library

3.3.5. Folders

Submitted Folder

This is a local folder, which store student submitted projects. This project synced with the Dropbox submitted projects folder.

The dir path config in application.properties files.

```
submitted_projects=E:\\Work\\Studi\\feedback_resource\\submitted_projects
```

Figure 3.26: Submitted folder configuration

Project Processing

This is a local folder that stores student projects while they are being tested. The dir path config in application.properties files.



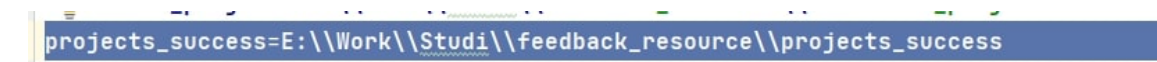
```
projects_processing=E:\\Work\\Studi\\feedback_resource\\projects_processing
```

Figure 3.27: Project Processing folder configuration

Project Success

This is a local folder that stores student projects that are tested and pass both the Junit test and PiTest coverage threshold.

The dir path config in application.properties files.



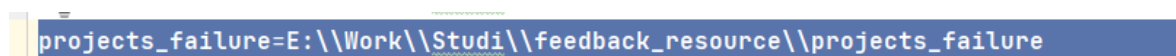
```
projects_success=E:\\Work\\Studi\\feedback_resource\\projects_success
```

Figure 3.28: Project Success folder configuration

Project Failure

This is a local folder that stores student projects that are tested and do not pass the Junit test or PiTest coverage threshold.

The dir path config in application.properties files.



```
projects_failure=E:\\Work\\Studi\\feedback_resource\\projects_failure
```

Figure 3.29: Project Failure folder configuration

3.3.6. Application properties file

Application.properties file is added in the project where the professor can set the threshold value of mutation coverage percentage.

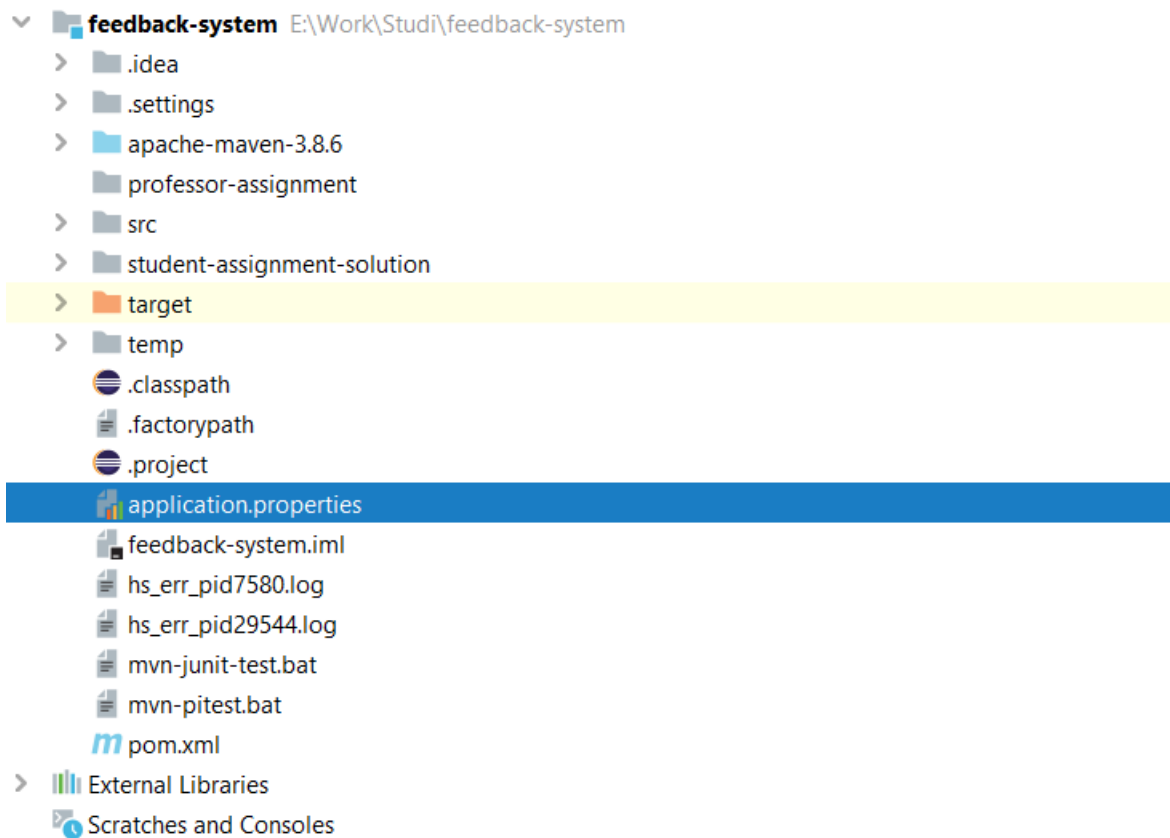


Figure 3.30: Application properties file

3.3.7. Resources

Feedback message.txt.

A template for a feedback message is found in the file `feedback message.txt`, which is kept in the resources folder. Using this template, a unique feedback message regarding the PiTest report's survivor mutants is created and delivered to the student as part of their feedback report.

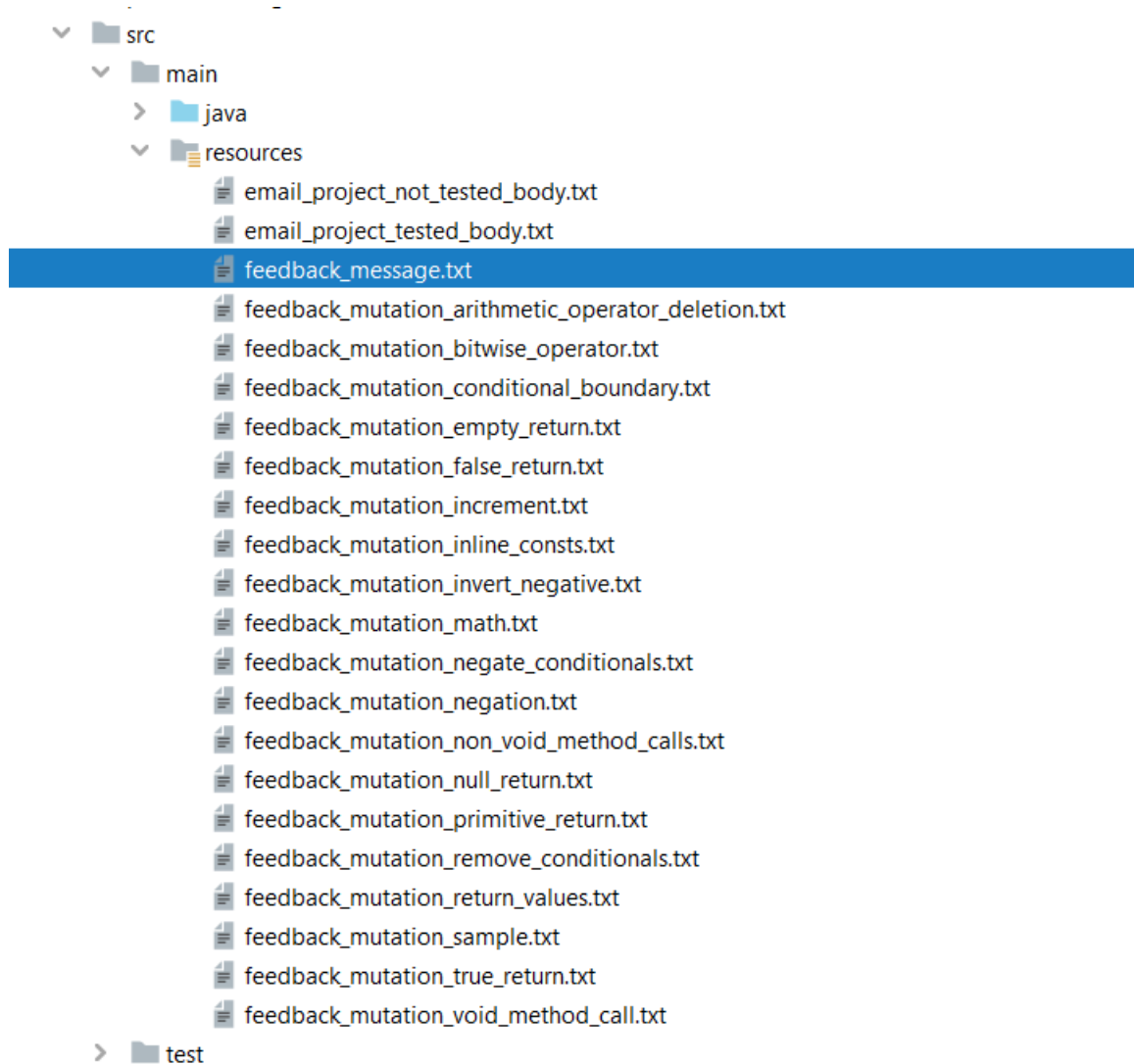


Figure 3.31: Feedback Message file

Feedback mutation sample.txt.

The feedback mutation sample.txt file is also included in the resources folder.

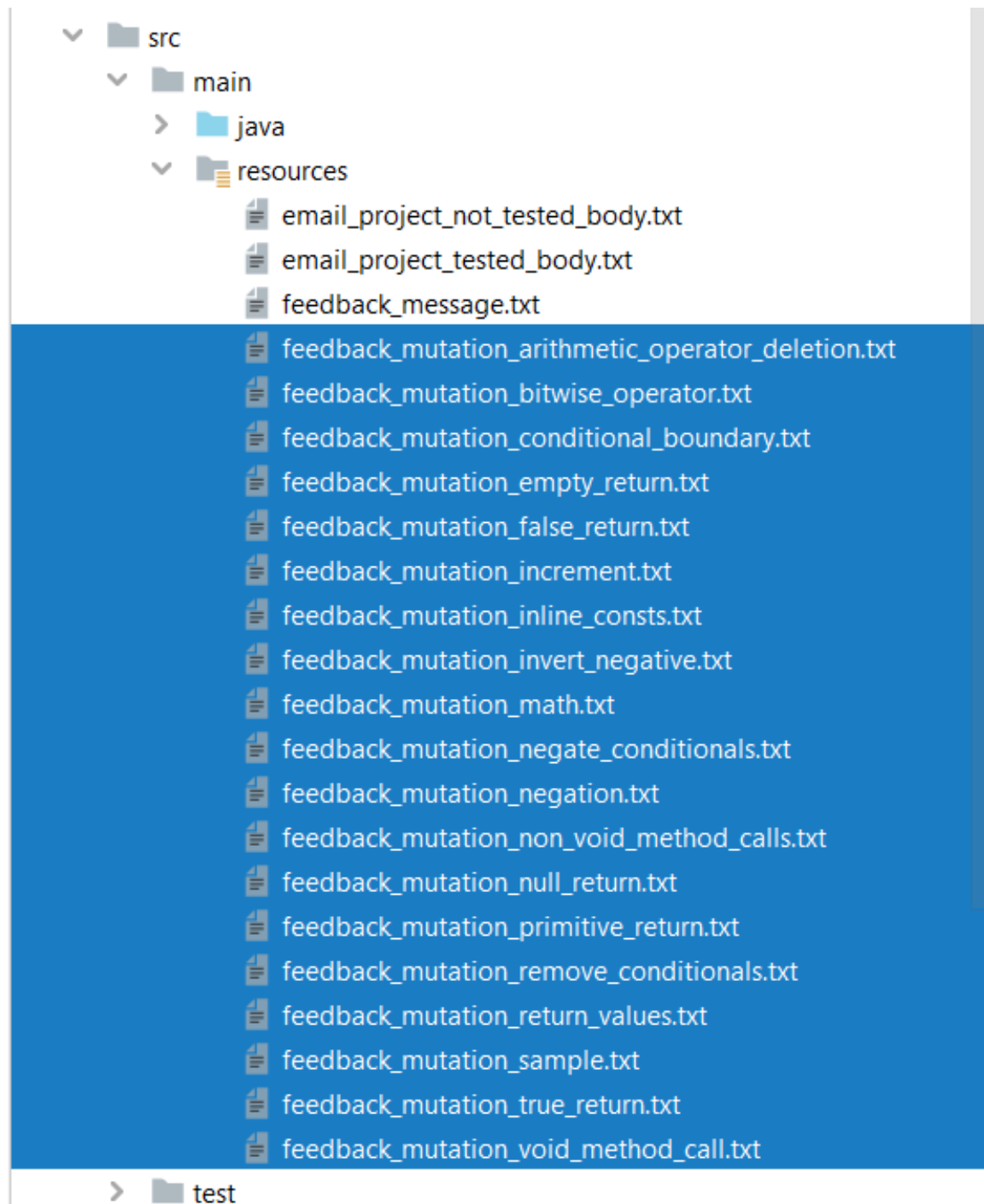


Figure 3.32: Mutation Samples file

3.3.8. Email project tested body.txt

The body of email notification to students when the submitted project is tested could be a failure or success to pass the testing threshold.

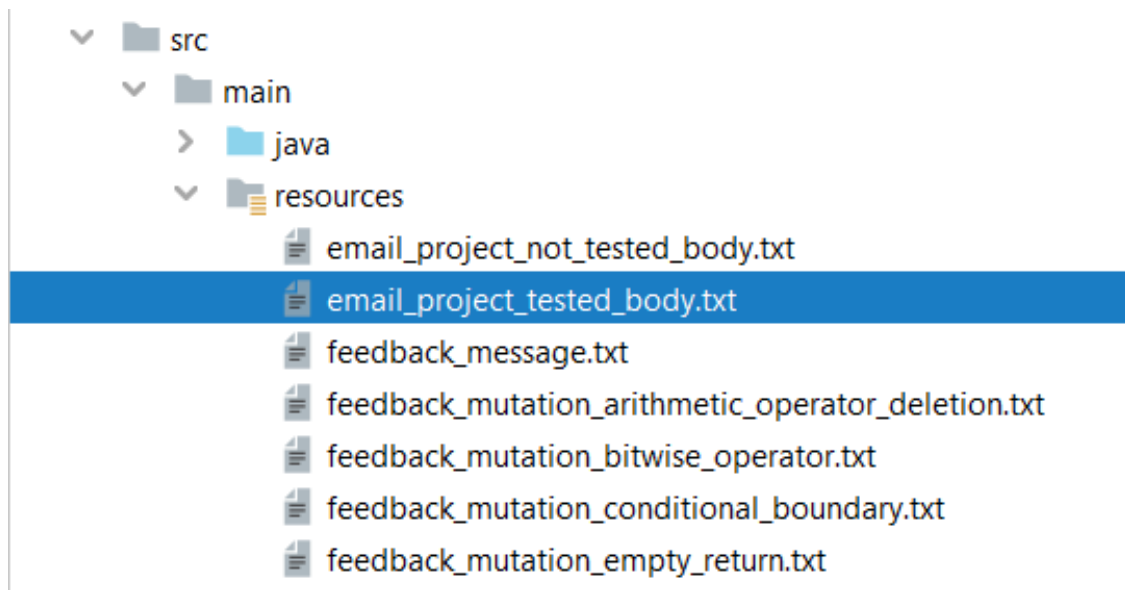


Figure 3.33: Email project tested body

Email project not tested body.txt

The body of email notification to students when the submitted project is not tested could be the wrong project format name, or the project does not have a pom.xml file.

Dear <StudentName>

I am pleased to inform you that your project has been tested. Depending on the results of the testing, I have the following feedback for you:

Body 1: Project Not Tested: The project had been scanned; however, it was not tested due to <the reason>. Please check the project in the folder 'project_not_tested' on Dropbox and resubmit it after you have fixed any issues.

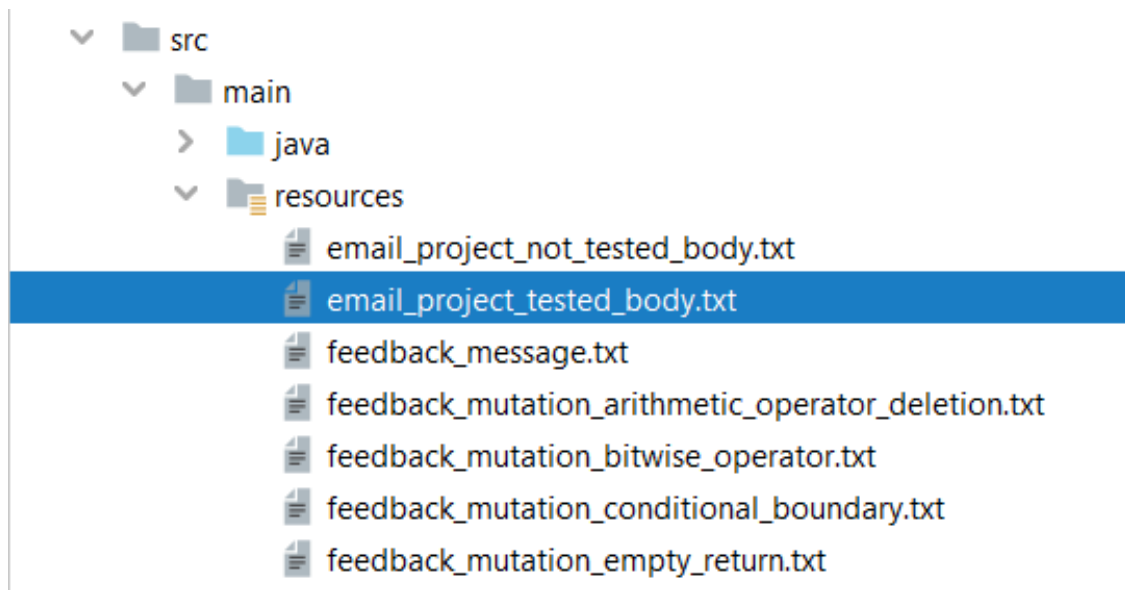


Figure 3.34: Email project_tested_body.txt

Body 2: Project Tested but Failed at Unit Testing: The project had been scanned and tested but there are some test cases that failed. <List test case failed>. Please update these test cases and resubmit them.

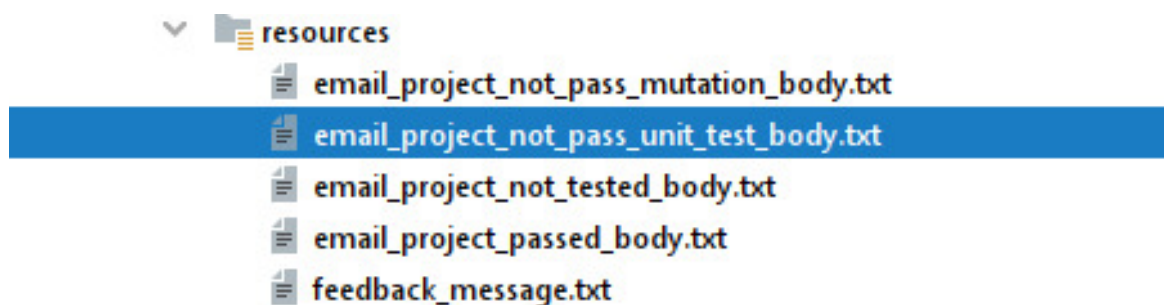


Figure 3.35: Email project_not_pass_unit_test_body.txt

Body 3: Project Tested but Failed at Mutation Testing: The project had been scanned and tested but the test case is not good enough. You need to improve the test case to make it better. Please read the attached file 'feedBackReport.txt' for further details regarding how to improve the test case.

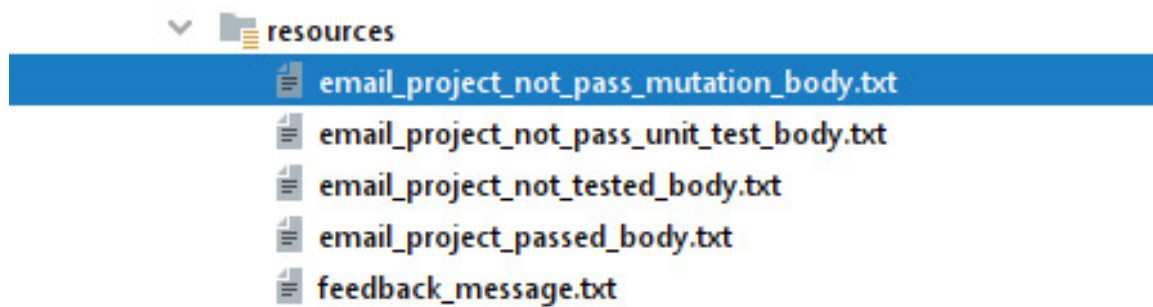


Figure 3.36: Email project_not_pass_mutation_body.txt

Body 4: Project Tested and Successful: Congratulations! The project has passed all checks and been successfully tested. Well done and keep up the good work! If you have any questions or would like further clarification, please do not hesitate to contact me directly or via email.

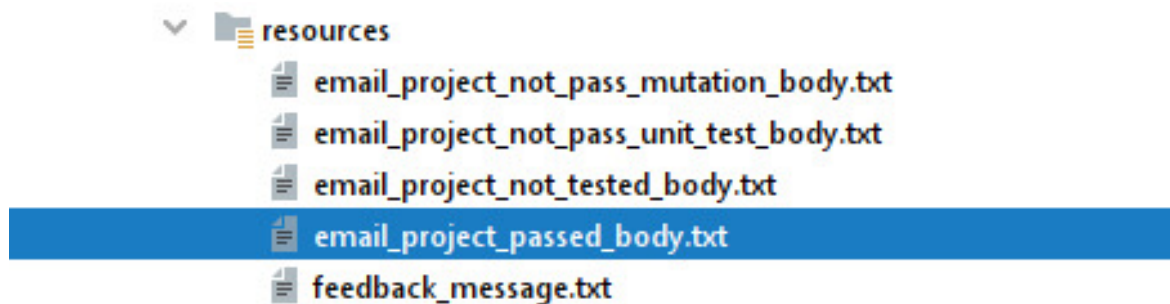


Figure 3.37: Email project_passed_body.txt

Regards,
[Professor Email Signature]

Project format name:

Mr.StudenName_ProjectName_Email_yyymmddHHMM

Mrs.StudenName_ProjectName_Email_yyymmddHHMM

Project name Restriction:

The length of the project name is can be only between 1 to 40 characters

Example of project name:

Mr.studiName_BankingApplicationSystem_anhnd6893@gmail.com_202301091803

3.3.9. Feedback report

PiTest generated report

After executing the PiTest, the mutation test report named “PiTest-reports” will be generated in the folder name “target” of the project being tested.

Example:

StudentBankingAppAssignment_kontaktlourenz@yahoo.de_202301070608\target\PiTest-reports

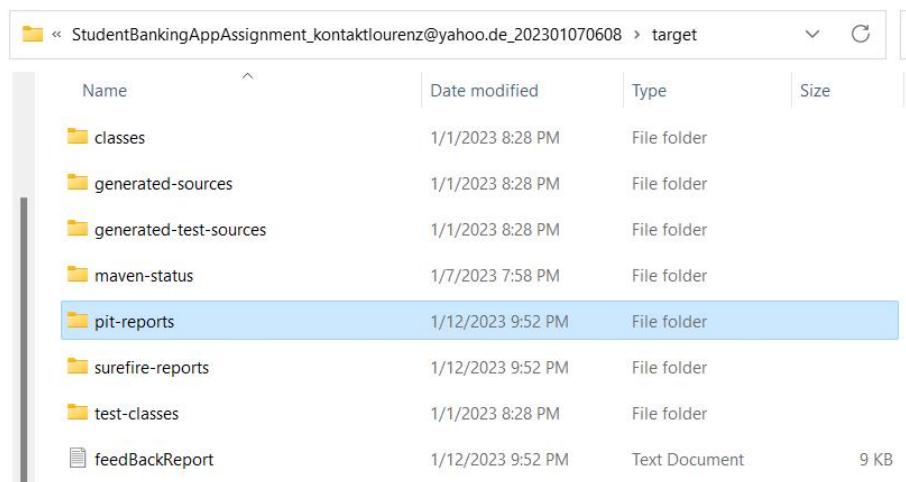
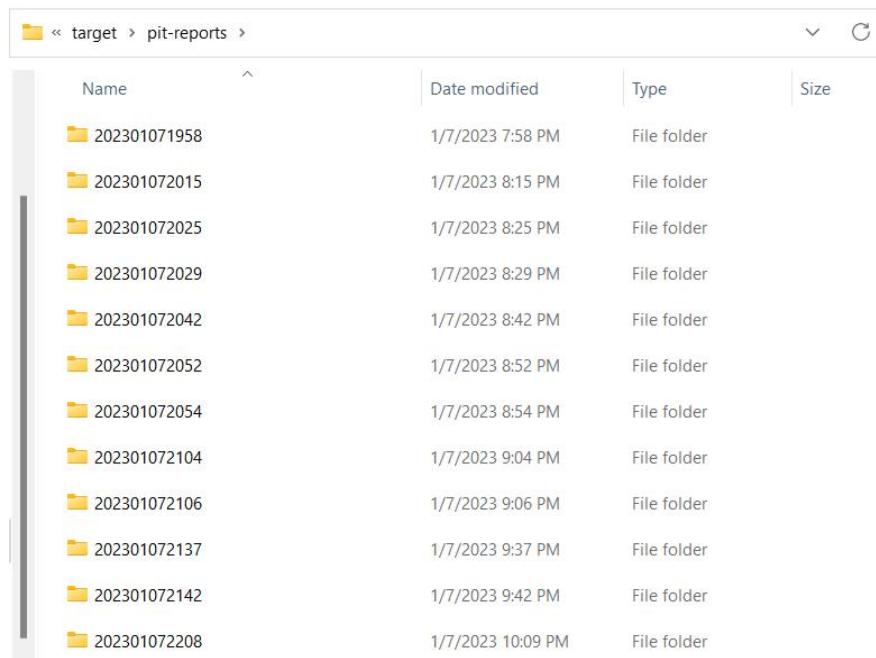


Figure 3.38: PiTest report location

It contains a list of mutation test history folders named in the format year-date-month-hour-mins.

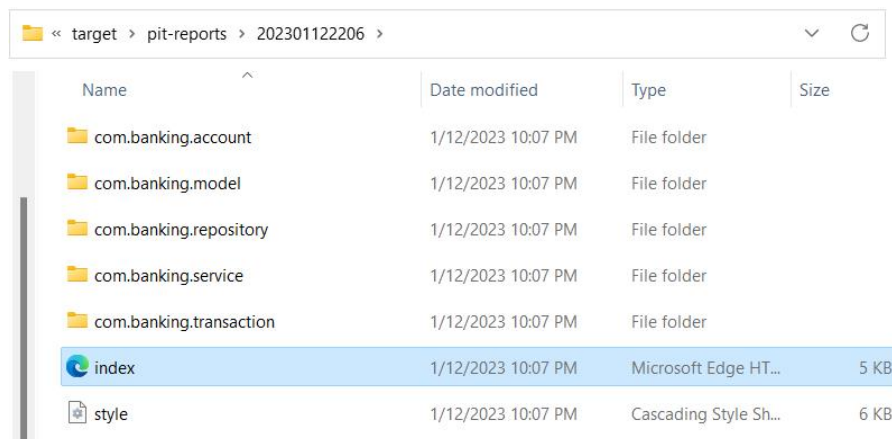
Example: 202301071958



Name	Date modified	Type	Size
202301071958	1/7/2023 7:58 PM	File folder	
202301072015	1/7/2023 8:15 PM	File folder	
202301072025	1/7/2023 8:25 PM	File folder	
202301072029	1/7/2023 8:29 PM	File folder	
202301072042	1/7/2023 8:42 PM	File folder	
202301072052	1/7/2023 8:52 PM	File folder	
202301072054	1/7/2023 8:54 PM	File folder	
202301072104	1/7/2023 9:04 PM	File folder	
202301072106	1/7/2023 9:06 PM	File folder	
202301072137	1/7/2023 9:37 PM	File folder	
202301072142	1/7/2023 9:42 PM	File folder	
202301072208	1/7/2023 10:09 PM	File folder	

Figure 3.39: PiTest report history

The detail of the PiTest is in the html file.



Name	Date modified	Type	Size
com.banking.account	1/12/2023 10:07 PM	File folder	
com.banking.model	1/12/2023 10:07 PM	File folder	
com.banking.repository	1/12/2023 10:07 PM	File folder	
com.banking.service	1/12/2023 10:07 PM	File folder	
com.banking.transaction	1/12/2023 10:07 PM	File folder	
index	1/12/2023 10:07 PM	Microsoft Edge HT...	5 KB
style	1/12/2023 10:07 PM	Cascading Style Sh...	6 KB

Figure 3.40: PiTest report detail location

AccountType.java

```
1 package com.banking.account;
2
3 /*
4  * This Enum class represents the type of account that a user can have.
5  */
6 public enum AccountType {
7
8     SAVINGS(100),
9     CURRENT(200),
10    SALARY(300),
11    LOAN(400);
12
13    private int accountTypeCode;
14
15    AccountType(int accountTypeCode) {
16
17        this.accountTypeCode = accountTypeCode;
18    }
19
20    public int getAccountTypeCode() {
21
221 return accountTypeCode;
23    }
24
25 }
```

Mutations

22 1. replaced int return with 0 for com/banking/account/AccountType::getAccountTypeCode → NO_COVERAGE

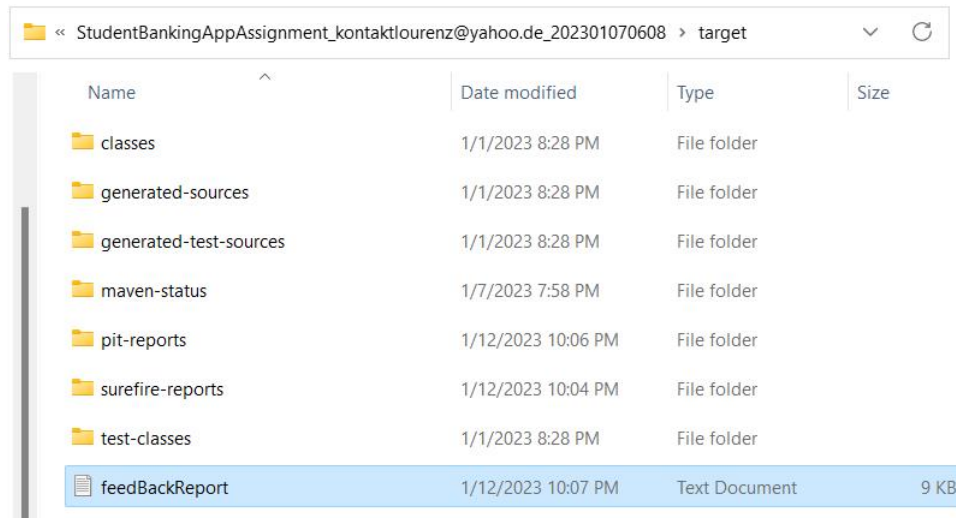
Figure 3.41: PiTest report detail

Feedback message report

The feedback message report file name “feedBack Report.txt” generated in the folder name “target” of the project being tested.

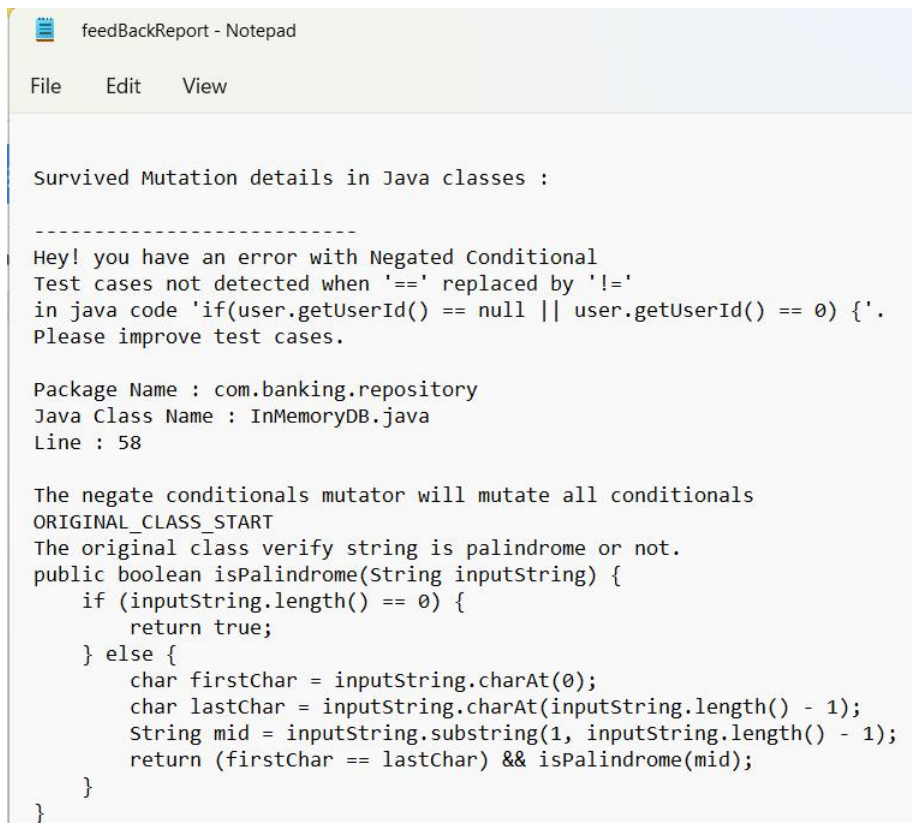
Example:

StudentBankingAppAssignment_kontaktlourenz@yahoo.de_202301070608\target\feedBackReport.tx



Name	Date modified	Type	Size
classes	1/1/2023 8:28 PM	File folder	
generated-sources	1/1/2023 8:28 PM	File folder	
generated-test-sources	1/1/2023 8:28 PM	File folder	
maven-status	1/7/2023 7:58 PM	File folder	
pit-reports	1/12/2023 10:06 PM	File folder	
surefire-reports	1/12/2023 10:04 PM	File folder	
test-classes	1/1/2023 8:28 PM	File folder	
feedBackReport	1/12/2023 10:07 PM	Text Document	9 KB

Figure 3.42: Feedback report file



```
feedBackReport - Notepad
File Edit View

Survived Mutation details in Java classes :

-----
Hey! you have an error with Negated Conditional
Test cases not detected when '==' replaced by '!='
in java code 'if(user.getUserId() == null || user.getUserId() == 0) {}'.
Please improve test cases.

Package Name : com.banking.repository
Java Class Name : InMemoryDB.java
Line : 58

The negate conditionals mutator will mutate all conditionals
ORIGINAL_CLASS_START
The original class verify string is palindrome or not.
public boolean isPalindrome(String inputString) {
    if (inputString.length() == 0) {
        return true;
    } else {
        char firstChar = inputString.charAt(0);
        char lastChar = inputString.charAt(inputString.length() - 1);
        String mid = inputString.substring(1, inputString.length() - 1);
        return (firstChar == lastChar) && isPalindrome(mid);
    }
}
```

Figure 3.43: A part of Feedback report content

4

EVALUATION

4.1. Overview:

The purpose of this section is to provide a broad overview of the major findings of the thesis, discuss their importance and answer the research questions raised in Chapter 1. The chapter evaluates and illustrate how the customized feedback system was developed and deployed to provide feedback hints to the students after submitting their JUnit Test solutions on the system. The PiTest is the tool that I used to generate mutants. The outcome of this study have shown that my newly developed customized feedback system was able to effectively provide meaningful and customizable feedback hints to early learners with a basic understanding of JUnit Testing skills. When my customized feedback system is compared to the PiTest feedback system, my feedback system offers better results since its feedback hints explain to the students where they went wrong and how to rectify their mistakes.

4.2. Evaluation Process

In this study, 45 students were contacted to participate in the research. Of them, 25 students showed interest to participate in the research while 20 students did not show interest. Of the 25 students that showed interest only 14 of them completed their assignments and submitted their solutions successfully while 11 students failed to submit their solutions. The student participants were provided with assignments of a Java Application named StudentBankingAppAssignment, which was a banking demo application, with functionalities to withdraw money, pay in money and check balance. The students who agreed to participate in the research were required to complete the assignments and submit their solutions to the proposed feedback system.

The students were then given more instructions, following which they were to carry out the tasks and submit their solutions. Additionally, I had to provide them instructions of how to name the project, which was provided in a folder that was stored in the drop box. The students were requested to provide a valid email address as the last piece of instruction. Projects processing, Projects failure, Projects Success, Projects Submitted, Projects Incorrect, project assignment was among the folders that could be found in the drop box. After then, the students had two weeks to finish the assignment and submit it via the feedback system.

I chose to give the students two weeks so they would have enough time to finish the task, considering that the students may have had other school courses with assignments that they had to complete.

The students that took part in the evaluation procedure ranged in age from 20 to 27 and were from different universities and colleges but they all belonged to the Computer Science Departments of their respective institutions. According to their feedback, the students who participated volunteered to participate to learn the fundamentals of JUnit testing as well as gain a better understanding of how advanced JUnit testing was carried out in programming assignments while others pointed out that their participation had to do with their passion in programming studies. Some of the students who participated had advanced levels of JUnit testing skills, while others were early learners. The students also had the choice of delivering their work as a Zip file or as a folder containing all the files required to complete their project.

Students received assistance by being given resources such as tutorial videos explaining to the related assignments, as well as samples of assignment solutions that had passed the basic stage of JUnit testing. This was done to make sure that the students understood what lay ahead of them as far as the assignment is concerned as well as what was expected of them when writing code for their project solutions. Additionally, the students were supposed to contact a given email address if they had any queries or encounter difficulties in the task they were assigned. The email was meant to be used for requesting help. It's also important to remember that the students that took part in this exercise were from colleges and universities located all over the continents of Africa and Europe.

The first step was for the students to access the assignment using a Dropbox folder labeled " project assignment." It is significant to note that not all of the students who were contacted agreed to take part in this evaluation procedure. The solutions to these students' assignments were not entered into the system because they either did not respond when assignments were given to them in the " project assignment" folder or did not follow the instructions that were given to them. While the solutions provided by a different group of students who took part in the assessment were successfully implemented, it was later discovered that these solutions contained errors that prevented the system from processing them further. As a result, these projects were moved into the "projects failure" folder.

The instructor, myself, determined the root causes of their errors and afterward analyzed the projects that were transferred to the "projects failure" folder in further detail. Another set of students also submitted valid solutions, however, some of these valid solutions did not follow the initial instructions, and as a consequence, they were unable to finish the procedure; as a result, the system transferred them to a folder named "projects incorrect format."

After the students successfully finished their projects, they had the option of providing their feedback or opinion based on their utilization of the developed feedback system as a whole and how well the assignments were handled. The feedback from the students was mainly positive, as they found that using the proposed feedback system had helped them to understand how advanced JUnit testing was carried out in programming assignments.

Many of the students commented on how useful the tutorials and sample solutions had been for giving them an understanding of what was expected when writing code for their own project solutions. Additionally, many of the participants expressed appreciation for being given two weeks to complete their assignments and submit their solutions, as it allowed them enough time to focus on other school coursework while still having enough time to finish their assignments.

In terms of negative feedback, some of the participants noted that they would have appreciated more detailed instructions regarding naming conventions for their projects. Others also pointed out that they would have liked more clarification on certain aspects of the assignment such as which libraries were required and what specific types of code snippets were needed for certain functions. Some students also noted that they would have liked more resources available in terms of tutorials or sample solutions.

Most of the participants found that using the proposed feedback system was helpful and provided them with a better understanding of how JUnit testing worked in programming assignments. The majority of comments related to issues with instructions or lack of resources, which could be addressed by providing additional guidance or making more resources available online.

I evaluated the feedback they provided that was saved in a Dropbox folder called "student feedback opinion." More so this students' feedback were written in text files. However, it is important to note that the students' comments and opinions were given anonymously because no personal information regarding the participants was required for this evaluation to avoid data privacy violation. Additionally, the main reason why the feedback was delivered in written text files is just for simplicity's sake. It was necessary to make it easy for students to provide their feedback of their experience after using the proposed feedback system.

4.3. Importance of the findings

The data collected from my proposed feedback system with mutation testing shows that it is both simple and provides meaningful feedback. I had carefully categorized and organized the codebase into packages and folders, which made the system easier to use and understand for tutors and professors. Additionally, I used Dropbox to store project files submitted by students, which increased the flexibility of the system.

The evaluation process revealed that the student participants responded positively to the system and found it easy to use while providing meaningful feedback. The students felt that they could finish their assignments quickly due to the detailed instructions provided by me as well as the tutorial videos which helped them understand better how JUnit testing was conducted in programming assignments. Furthermore, they also appreciated the fact that they could submit their solutions as a Zip file or folder containing all the files required for their assignment which made it easier for them to complete their tasks.

The advantages of using JUnit tests in code were also demonstrated during this evaluation process. Firstly, writing unit tests helps create a robust system capable of responding gracefully

to unforeseen events and user errors. Secondly, JUnit tests can help identify coding bugs early in the development process and provide meaningful feedback that helps developers fix these issues quickly and accurately. Thirdly, unit tests can be used for refactoring code, as they will quickly identify any unintended changes that may occur due to design modifications. Finally, JUnit tests provide a layer of documentation for the codebase making it easier for developers to understand the structure and purpose of each component. Having a good test coverage also ensures improved quality of software by helping us validate if our application is free of defects and if it is working as per our expectations. Furthermore, it makes the codebase easy to maintain and reusable so anyone can reference them and execute the test cases in the future.

Overall, my data shows that my proposed feedback system was indeed simple and provided meaningful feedback which helped students improve their programming skills quickly and accurately. The use of JUnit testing was key in this process as it enabled me to detect bugs early on in the development process and ensured that my system was robust enough to handle unforeseen events or user errors. All these factors combined resulted in meaningful feedback being generated for programming assignments with mutation testing, thus providing a better understanding result when compared to traditional methods such as manual reviews or peer grading systems.

4.4. Research questions:

1. What type of auto-generated feedback hints is useful for early learners of OOP with basic knowledge of Unit Testing? The first research question focused on determining what type of auto-generated feedback hints would be useful for early learners of OOP with basic knowledge of Unit Testing. The results showed that providing customized feedback hints that were made easy for students to understand and tailored to the individual student's solutions was beneficial for early learners. Furthermore, utilizing mutation testing tools such as PiTest can help improve the test suites used to evaluate student solutions, leading to more accurate assessment results.
2. How can the proposed feedback system be fully automated to generate meaningful feedback hints to early learners of OOP with basic knowledge of JUnit testing? The second research question focused on how the proposed feedback system could be fully automated to generate meaningful feedback hints to early learners of OOP with basic knowledge of JUnit testing. The results indicated that the newly developed feedback system was able to effectively provide meaningful and customizable feedback hints to early learners of Object-Oriented Programming. The feedback system was developed using the PiTest mutation testing tool, which enabled the evaluation of the quality of the student's submissions. Through this system, students were able to submit their solutions and receive clear and concise feedback hints that were much better than what was generated by PiTest alone.

5

Discussion and Future Work

5.1. Summary

The results of this research indicate that providing customized feedback to early learners of Object-Oriented Programming is beneficial in several ways. Firstly, it can help them identify errors in their code more quickly. Secondly, it provides insight into potential areas of improvement in their solutions, which can help them become better in writing JUnit test cases in the long run. Lastly, utilizing mutation testing tools such as PiTest is helpful to analyze the students test suites and to evaluate student solutions, leading to more accurate assessment results.

5.2. Interpretations

Looking at the customized feedback system, the evaluation has shown that giving customized feedback hints to students is important as it helps them identify errors in their code and rectify them. In the first interaction of the system, the PiTest tool has come out as an outstanding tool that is compatible with this system because it ensures the quality of the student software testing suite and also efficient in generating mutants.

Generating feedback is a useful way to point out errors in the students' solutions and emphasizes the importance of mutation testing to programming students. It is therefore a valuable technique, and relatively easy to implement using existing test frameworks. Most tools that use automated testing support the PiTest tool, because black-box testing does not require using a specific algorithm.

The only aspect that matters is whether the output meets the requirements of the exercise. However, only giving test-based feedback will not in all cases help a student fix an incorrect program. To help a student improve their JUnit testing skills, the feedback given should explain why the test failed, how the code was supposed to be, and suggestions on how to re-edit the code. Feedback on how to proceed is necessary to both fixing problems and progress toward a solution when the students are facing issues. This type of feedback is mostly seen in the form of error correction feedback hints and much less as hints on task-processing steps as well as program improvements.

The findings from this research suggest that providing customizable feedback hints is beneficial for early learners of Object-Oriented Programming because it helps them identify their errors and fix them more quickly and easily. It also provides insight into potential areas for improvement in their solutions. Furthermore, utilizing mutation testing tools such as PiTest can help improve the test suites used to evaluate student solutions and lead to more accurate assessment results. Thus, this type of feedback system could be useful for helping students learn and improve their skills over time. Tests suits that passed mutation testing means they are good.

The system offers the professor and tutor options to customize the feedback hints using text files format. Using text file format, the professor can specify custom feedback hints and the accepted threshold that the student must obtain to be considered as passed. There is a vast number of studies that identify and classify difficulties and errors that early learners encounter in their learning process. In my case, the customizable feedback hints had better level of feedback satisfaction compared to the Pitest generated feedback.

5.3. Advantages

The advantages of this study are significant because they suggest that providing meaningful and customizable feedback hints is an effective way to help early learners with basic knowledge of OOP improve their JUnit testing skills. This type of system would be especially useful for professors or tutors who are teaching large classes or dealing with a large volume of assignments regularly and due to its automated nature, which reduces the amount of time spent giving individualized feedback for each student's assignment solution.

Many of the previous feedback systems have only been able to deliver generalized feedback. However, my customized feedback system goes further to allowing the professor or tutor give a customized feedback hint that early learners with basic knowledge JUnit testing skills can understand. If students want to learn from their mistakes, they should familiarize themselves with using my feedback system. The use of this system enables lecturers to focus their attention to the students whose code failed. By doing so they can engage the students in person and help them identify their weaknesses and improve on them.

5.4. Limitations

The evaluation process used in this study is subject to several limitations that threaten its validity. The first issue relates to the accuracy and meaningfulness of the variables being measured. In this study, the feedback system was tested on a small sample size of 45 students, 25 of whom agreed to participate. This limited sample size may not accurately represent the wider population, as it is impossible to guarantee that the participants were representative of the wider student population in terms of age, experience level and other factors. Furthermore, the assignment given to the participants was relatively simple and straightforward; thus, there may be no way to extrapolate any findings from this evaluation process to more complex programming assignments.

Another issue is whether or not any changes observed in performance are attributable solely to the feedback system. As mentioned before, some students failed to submit solutions due to a lack of understanding or technical difficulties; such factors could have had an impact on their performance and should be accounted for when analyzing results. Additionally, the instructor's involvement in providing resources such as tutorial videos and sample solutions could have inadvertently influenced the students' performance; thus, it would be advisable for future research projects to look into different approaches for providing assistance without having an effect on student performance.

Finally, the last limitation deals with how generalizable results from this study are for similar studies in different contexts. The participants in this study were from universities located all over Europe and Africa, so it would be difficult to draw conclusions from this study about how well a similar feedback system would perform at institutions located elsewhere. Furthermore, as noted earlier, some students found certain aspects of the assignment difficult or overly time-consuming; thus, any conclusions drawn from this study may not apply when dealing with more complex assignments at other institutions.

In conclusion, while this evaluation process has provided valuable insights into how well a feedback system can improve student programming performance through mutation testing techniques, further research is necessary before these findings can be generalized across different contexts.

5.5. Conclusion

The research on feedback systems has demonstrated that providing relevant and customizable feedback hints is a successful strategy for assisting early learners in learning and improving their JUnit testing skills over time. The evaluation showed that the structure of the customizable feedback hints was simple, which enabled the students to understand it easily. This implies that the system will so be great for lecturers who have to deal with a large number of students assignments. Although early learners are the target group, the system can also generate feedback for more experienced programmers because the feedback hints are customizable.

The feedback system was organized into categories and arranged into basic folders for easy understanding and scalability. Descriptive class and method names were used to enable the readability of the source code by other Java developers. This approach facilitates future expansions while ensuring clarity of the system design.

Tracking activity of students using the system revealed that those who got regular feedback hints submitted more solutions than those who did not get any hints. In addition, a comparison between the time taken by feedback systems and manual feedback showed a significant difference in favor of automated evaluation and giving of feedback. The data gathered from my feedback system suggests that it is indeed simple and provides meaningful feedback hints that aid students in improving their programming skills quickly. Furthermore, the feedback hints are customizable, making the system even more effective in helping students reach their desired proficiency level.

In conclusion, this research has demonstrated that providing relevant and customizable feedback hints is an effective strategy for assisting early learners in learning and improving their JUnit testing skills over time. The development of feedback systems opened the door for significant programming improvements in the way JUnit testing concepts were presented to

early learners of OOP and how the customized feedback hints assisted them in sharpening their skills.

The use of feedback systems is generally perceived to be excellent in giving customized hints on JUnit tests; however, further research should look more closely at how historical information about student solutions may be used to enhance such suggestions as well as explore new technologies and designs used for creating future automated evaluation systems with multiple feedback hints per mutation testing process.

5.6. Future Work

Based on these findings, it is recommended that further analyses could be conducted on how automated feedback systems affect students at different levels or with different types of programming backgrounds before any definitive conclusions can be drawn about its effectiveness overall across all types of programming courses or assignments. I have tried to locate as many various references that will support my arguments as feasible to make my system as dependable as possible. My thesis' credibility is founded on present knowledge and actuality.

Additionally, future work shall involve adding a user interface so that the professor does not influence the feedback hints by adjusting the customizable feedback hints and thresholds more easily. The system shall also generate feedback hints that are auto-edited instead of having the professor pre-written on text files which is tiresome and time-consuming.

References

- [1] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, 4-es, 2005.
- [2] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys (CSUR)*, vol. 37, no. 2, pp. 83–137, 2005.
- [3] M. McCracken *et al.*, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," in *Working group reports from ITiCSE on Innovation and technology in computer science education*, 2001, pp. 125–180.
- [4] M. E. Caspersen and J. Bennedsen, "Instructional design of a programming course: A learning theoretic approach," in *Proceedings of the third international workshop on Computing education research*, 2007, pp. 111–122.
- [5] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming," *AcM SIGcSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007.
- [6] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 491–502.
- [7] V. J. Shute, "Focus on formative feedback," *Review of educational research*, vol. 78, no. 1, pp. 153–189, 2008.
- [8] J. Hattie and H. Timperley, "The power of feedback," *Review of educational research*, vol. 77, no. 1, pp. 81–112, 2007.
- [9] D. Boud and E. Molloy, "What is the problem with feedback?" In *Feedback in higher and professional education*, Routledge, 2012, pp. 1–10.
- [10] V. J. Shute, "Focus on formative feedback," *Review of educational research*, vol. 78, no. 1, pp. 153–189, 2008.
- [11] D. C. Merrill, B. J. Reiser, M. Ranney, and J. G. Trafton, "Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems," *The Journal of the Learning Sciences*, vol. 2, no. 3, pp. 277–305, 1992.
- [12] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, 2004, pp. 317–325.
- [13] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 147–158.

- [14] J. Mořucha and B. Rossi, "Is mutation testing ready to be adopted industry-wide?" In *International Conference on Product-Focused Software Process Improvement*, Springer, 2016, pp. 217–232.
- [15] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.
- [16] A. M. Kazerouni, J. C. Davis, A. Basak, C. A. Shaffer, F. Servant, and S. H. Edwards, "Fast and accurate incremental feedback for students' software tests using selective mutation analysis," *Journal of Systems and Software*, vol. 175, p. 110905, 2021.
- [17] M. A. Cachia, M. Micallef, and C. Colombo, "Towards incremental mutation testing," *Electronic Notes in Theoretical Computer Science*, vol. 294, pp. 2–11, 2013.
- [18] G. Haldeman, "Automated feedback generation for programming assignments," Ph.D. dissertation, Rutgers The State University of New Jersey, School of Graduate Studies, 2021.
- [19] A. Varma, S. Tripathi, and R. Prasad, *Plant microbe interface*. Springer, 2019.
- [20] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *International Symposium on Search Based Software Engineering*, Springer, 2020, pp. 9–24.
- [21] "Calr mutation testing.com," in *Accessed: December 30, 2022*, Springer, 2020.
- [22] P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, and O. Nierstrasz, "A decade of code comment quality assessment: A systematic literature review," *Journal of Systems and Software*, p. 111515, 2022.
- [23] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson, "Providing test quality feedback using static source code and automatic test suite metrics," in *16th IEEE international symposium on software reliability engineering (ISSRE'05)*, IEEE, 2005, 10–pp.
- [24] M. Pike, B. G. Lee, and D. Towey, "Taffies: Tailored automated feedback framework for developing integrated and extensible feedback systems," *SN Computer Science*, vol. 3, no. 2, pp. 1–8, 2022.
- [25] K. Buffardi and S. H. Edwards, "Impacts of adaptive feedback on teaching test-driven development," in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 293–298.
- [26] A. Francisco, H. Truong, A. Khosrowpour, J. E. Taylor, and N. Mohammadi, "Occupant perceptions of building information model-based energy visualizations in eco-feedback systems," *Applied Energy*, vol. 221, pp. 220–228, 2018.
- [27] N. Soni, K. Jyoti, U. K. Jain, A. Katyal, R. Chandra, and J. Madan, "Noscapinoids bearing silver nanocrystals augmented drug delivery, cytotoxicity, apoptosis and cellular uptake in b16f1, mouse melanoma skin cancer cells," *Biomedicine & pharmacotherapy*, vol. 90, pp. 906–913, 2017.
- [28] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

- [29] Q. V. Nguyen and L. Madeyski, "Empirical evaluation of multiobjective optimization algorithms searching for higher order mutants," *Cybernetics and Systems*, vol. 47, no. 1-2, pp. 48–68, 2016.
- [30] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports quality assurance processes," *Software Testing, Verification and Reliability*, vol. 28, no. 6, e1675, 2018.
- [31] Q. V. Nguyen and L. Madeyski, "Problems of mutation testing and higher order mutation testing," in *Advanced computational methods for knowledge engineering*, Springer, 2014, pp. 157–172.
- [32] A. Roman and M. Mnich, "Test-driven development with mutation testing—an experimental study," *Software Quality Journal*, vol. 29, no. 1, pp. 1–38, 2021.
- [33] F. Toure, M. Badri, and L. Lamontagne, "Towards a unified metrics suite for junit test cases.," in *SEKE*, 2014, pp. 115–120.
- [34] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica*, vol. 35, no. 3, 2011.
- [35] K. Liu *et al.*, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.
- [36] A. Jaffari and C.-J. Yoo, "An experimental investigation into data flow annotated-activity diagram-based testing," *Journal of Computing Science and Engineering*, vol. 13, no. 3, pp. 107–123, 2019.
- [37] J. Fang, "A critical review of five machine learning-based algorithms for predicting protein stability changes upon mutation," *Briefings in bioinformatics*, vol. 21, no. 4, pp. 1285–1292, 2020.
- [38] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: An analysis and survey," in *Advances in Computers*, vol. 112, Elsevier, 2019, pp. 275–378.
- [39] M. B. Bashir and A. Nadeem, "An evolutionary mutation testing system for java programs: Emujava," in *Intelligent Computing-Proceedings of the Computing Conference*, Springer, 2019, pp. 847–865.
- [40] H. Chen *et al.*, "{Muzz}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2325–2342.
- [41] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, 2019.
- [42] S. Peldszus, "Model-driven development of evolving secure software systems.," in *Software Engineering (Workshops)*, 2020.

- [43] S. Dick, S. Schulz, and C. Bockisch, "A study on the quality mindedness of students," *Software Engineering im Unterricht der Hochschulen (SEUH 2022)*, 2022.
- [44] C. Desai, D. Janzen, and K. Savage, "A survey of evidence for test-driven development in academia," *ACM SIGCSE Bulletin*, vol. 40, no. 2, pp. 97–101, 2008.
- [45] S. H. Edwards, "Rethinking computer science education from a test-first perspective," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp. 148–155.
- [46] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," in *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 2004, pp. 26–30.
- [47] C. Matthies, A. Treffer, and M. Uflacker, "Prof. ci: Employing continuous integration services and github workflows to teach test-driven development," in *2017 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2017, pp. 1–8.
- [48] F. Toure, M. Badri, and L. Lamontagne, "Towards a unified metrics suite for junit test cases.," in *SEKE*, 2014, pp. 115–120.
- [49] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica*, vol. 35, no. 3, 2011.
- [50] K. Liu *et al.*, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.
- [51] A. Jaffari and C.-J. Yoo, "An experimental investigation into data flow annotated-activity diagram-based testing," *Journal of Computing Science and Engineering*, vol. 13, no. 3, pp. 107–123, 2019.
- [52] J. Fang, "A critical review of five machine learning-based algorithms for predicting protein stability changes upon mutation," *Briefings in bioinformatics*, vol. 21, no. 4, pp. 1285–1292, 2020.
- [53] J. Hattie and H. Timperley, "The power of feedback," *Review of educational research*, vol. 77, no. 1, pp. 81–112, 2007.
- [54] V. J. Shute, "Focus on formative feedback," *Review of educational research*, vol. 78, no. 1, pp. 153–189, 2008.
- [55] D. Whitelock, A. Twiner, J. T. Richardson, D. Field, and S. Pulman, "Openessayist: A supply and demand learning analytics tool for drafting academic essays," in *Proceedings of the fifth international conference on learning analytics and knowledge*, 2015, pp. 208–212.
- [56] T. Barnes and J. Stamper, "Automatic hint generation for logic proof tutoring using historical data," *Journal of Educational Technology & Society*, vol. 13, no. 1, pp. 3–12, 2010.
- [57] D. Whitelock, "Advice for action with automatic feedback systems," *Software Data Engineering for Network eLearning Environments*, pp. 139–160, 2018.

- [58] E. A. Özbek, "Plagiarism detection services for formative feedback and assessment: Example of turnitin," *Journal of Educational and Instructional Studies in the World*, vol. 6, no. 3, pp. 64–72, 2016.
- [59] A. Lachner, C. Burkhart, and M. Nückles, "Formative computer-based feedback in the university classroom: Specific concept maps scaffold students' writing," *Computers in Human Behavior*, vol. 72, pp. 459–469, 2017.
- [60] N. Madnani and A. Cahill, "Automated scoring: Beyond natural language processing," in *Proceedings of the 27th International Conference on Computational Linguistics*, 2018, pp. 1099–1109.
- [61] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice programming students," *Computer Science Education*, vol. 26, no. 2-3, pp. 148–175, 2016.
- [62] B. A. Becker, K. Goslin, and G. Glanville, "The effects of enhanced compiler error messages on a syntax error debugging test," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 640–645.
- [63] S. H. Edwards and K. P. Murali, "Codeworkout: Short programming exercises with built-in data collection," in *Proceedings of the 2017 ACM conference on innovation and technology in computer science education*, 2017, pp. 188–193.