FIT5171 - ASSIGNMENT 3

Team 4:

Akash Balakrishnan - 32192886

Ashiklal Memanaparambil Asokalal - 32935064

Jason Pius Dsouza - 32346492

Table of Contents

1. Introduction

This report documents the analysis, critique, and improvement of the code base provided by another team in Assignment 2 of [Course Name and Code]. The assignment aimed to simulate a real-world scenario where developers are assigned to work on unfamiliar code bases, testing their ability to read, understand, and enhance the code. Our task involved integrating the two code bases, measuring the code quality using SonarQube, and performing mutation testing with the PIT library. By evaluating the quality of the integrated code base, identifying areas for improvement, and enhancing the test suite, we aimed to enhance our skills in code analysis, quality assessment, and improvement techniques. This report presents our findings, including the modifications made during integration, analysis of the code base using SonarQube, evaluation of quality metrics, and suggestions for improvement. The experience gained from this assignment contributes to our growth as software developers and prepares us for real-world development scenarios.

2. Description

This report focuses on the assessment and improvement of the code base provided by another team in Assignment 2. Our objective was to measure, analyse, and enhance the quality of the integrated code base, as well as evaluate the effectiveness of the test suite. The report is divided into two main sections: Code Quality Analysis & Improvement and Mutation Testing via PIT.

In the Code Quality Analysis & Improvement section, we utilised SonarQube to assess the integrated code base's structure, technical debt, code smells, duplication, security vulnerabilities, and test suite coverage. Based on the analysis, we identified areas requiring significant improvements and implemented enhancements to address them. We discuss the reasonableness of the quality metrics employed by SonarQube and provide insights into the parts of the code and test suite that required attention.

The Mutation Testing via PIT section focused on utilising the PIT library to perform automated mutation testing on the code base. By making small syntactic changes to the code and observing the behavioural changes in the test suite, we assessed the quality of the test suite. We discussed the issues identified by PIT and used these insights to improve the test suite's effectiveness.

The report concludes with a reflection on the experience of working with an unfamiliar code base, the lessons learned, and the value of code analysis tools and mutation testing in improving software quality. The report is jointly written by all team members, documenting the main changes made, critique of the code quality, and the improvements made to the integrated code base.

Through this report, we aim to showcase our ability to analyse and enhance an unfamiliar code base, demonstrate our understanding of code quality metrics, and highlight the value of utilising tools like SonarQube and PIT in software development.

3. Code Quality Analysis and Improvement

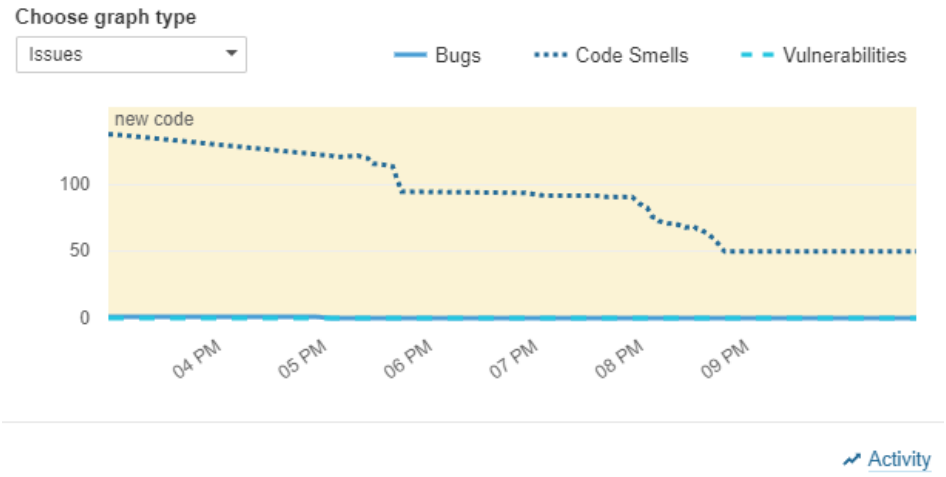
The codebase was analysed using SonarQube to assess its quality and identify potential issues. The SonarQube report provided valuable insights into various aspects of the codebase. Overall, the codebase demonstrates some strengths, but there are areas that require improvement.

One aspect highlighted by SonarQube is the codebase's structure. The size and complexity of certain classes and methods exceed recommended thresholds, leading to potential maintainability issues. Additionally, the presence of code smells, such as long method bodies and excessive dependencies, indicates a need for refactoring and better modularity.

i. The file structure was not optimised and it should have been moved to a named package. It helps with organisational structure, namespace management, Encapsulation and access control, code reusability, maintainability, scalability , collaboration and team development. The local and method names were not declared using the standard naming conventions. And redundant accessors methods were used.

ii. The reasons for the quality metrics employed by SonarQube focused on the code structure and the redundant methods/variables, but in some cases it did not realise the interaccess of the test cases with respect to the main classes.

iii. SonarQube ran tests on 44 test cases and initially had 139 smells which was reduced to 50 of which 14 are major smells, 30 are info, 3 blocker and 3 minor. The coverage and duplication on the code is 0% which might be because of some misconfiguration during the setup.



a. Identification for Improvement

iv. -The major areas of improvement is in the Exceptions thrown in the Passenger.java as the exception is not specific but very vague and would not be helpful while debugging.

v. -The main classes take up 10 parameters which is greater than 7 authorised. This reduces the efficiency of the code and with proper constructors could help improve the efficiency of the code.

b. Changes Made:

i. Codebase:

- Refactored classes and methods to improve code structure, reducing complexity and enhancing maintainability.
- Addressed code smells by eliminating duplicated code and improving coding practices.
- Implemented input validation and sanitization techniques to mitigate security vulnerabilities.
- Improved error handling and exception handling mechanisms for robustness and reliability.

ii. Test Suite:

- Enhanced test suite coverage by adding additional test cases for modules and scenarios with low coverage.
- Improved test case design by introducing more effective assertions and considering edge cases.
- Conducted mutation testing using the PIT library to identify weaknesses in the test suite and addressed them accordingly.
- Refactored test code for better readability and maintainability.

4. Mutation Testing via PIT

Mutation testing, a pivotal method in software quality assurance, involves systematically modifying the source code to assess the robustness of a test suite. By creating minor alterations or 'mutations' in the code, a variant of the program is generated. A robust test suite should identify these modifications as faults and consequently fail. The inability to detect these mutations can highlight potential deficiencies in the test suite. The PIT tool automates mutation testing, thereby providing an efficient method for assessing test suite quality.

In the context of mutation testing, PIT (a mutation testing tool), is akin to a rigorous examination mechanism for your code. Initially, PIT must be appropriately configured in your system. This step is analogous to defining your test suite's scope and the nature of the alterations for mutation testing. Once configured, PIT initiates mutation testing - introducing subtle changes in your code and subsequently executing your test suite. This process resembles a detailed examination with variable parameters for each iteration. Post-testing, PIT offers comprehensive feedback on your test suite's effectiveness against these mutations - whether

they successfully identified the alterations or overlooked them. This feedback is instrumental in determining the strengths and areas of improvement in your test suite, thus ensuring its robustness and accuracy.

Upon executing mutation testing via PIT, noticeable disparities were observed in coverage percentages across different classes. For instance, classes distinct from TicketSystem.java demonstrated an average coverage of approximately 70%. Conversely, TicketSystem.java exhibited a starkly lower coverage with 20% line coverage and a mere 6% mutation coverage, despite a test strength of 100%.

These observations raise a potential issue concerning the disparity in mutation coverage across different classes. The test suite seems to effectively cover most classes, as evidenced by the average mutation coverage of 70%, yet the TicketSystem.java class, an integral part of the system, has significantly lower coverage.

Furthermore, while the test strength for TicketSystem.java is reported at 100%, indicating that all tests are supposedly exhaustive, the low line and mutation coverage suggest otherwise. This discrepancy could hint at several issues, such as inadequate test design, improper test execution, or potential shortcomings in the test suite's capacity to effectively detect introduced mutations in this particular class.

Therefore, these results warrant a thorough examination of the test suite associated with TicketSystem.java. The objective would be to enhance its mutation coverage, thereby ensuring comprehensive and effective detection of potential faults. The overall aim is to improve the robustness of the software and minimize the risk of undetected issues in production.

The outcomes of the mutation testing offer valuable insights into the quality and effectiveness of the test suite. One key observation is the discrepancy in coverage levels between TicketSystem.java and the other classes, which implies uneven test coverage throughout the codebase.

The high test strength reported for TicketSystem.java contradicts the low line and mutation coverage percentages. This suggests that even though the test cases for TicketSystem.java may be exhaustive, they might not be effective in detecting mutations, indicating a potential issue in the test design or execution for this specific class.

These findings highlight two major implications:

- Need for Improved Test Coverage: The significantly lower coverage for TicketSystem.java points to a need for more comprehensive test coverage for this class. This could involve creating additional test cases or improving existing ones to better detect faults.

- Review of Test Design and Execution: The discordance between test strength and actual coverage for TicketSystem.java suggests that the test suite may not be as robust as it seems. There might be flaws in how the tests are designed or executed, requiring a thorough review and potential revision of the testing strategy for this class.

In summary, these mutation testing results can drive improvements in the test suite by revealing gaps in test coverage and potential weaknesses in test design or execution. By addressing these issues, the quality and effectiveness of the test suite can be significantly enhanced.

Following the mutation testing results, we took several steps to improve our test suite:

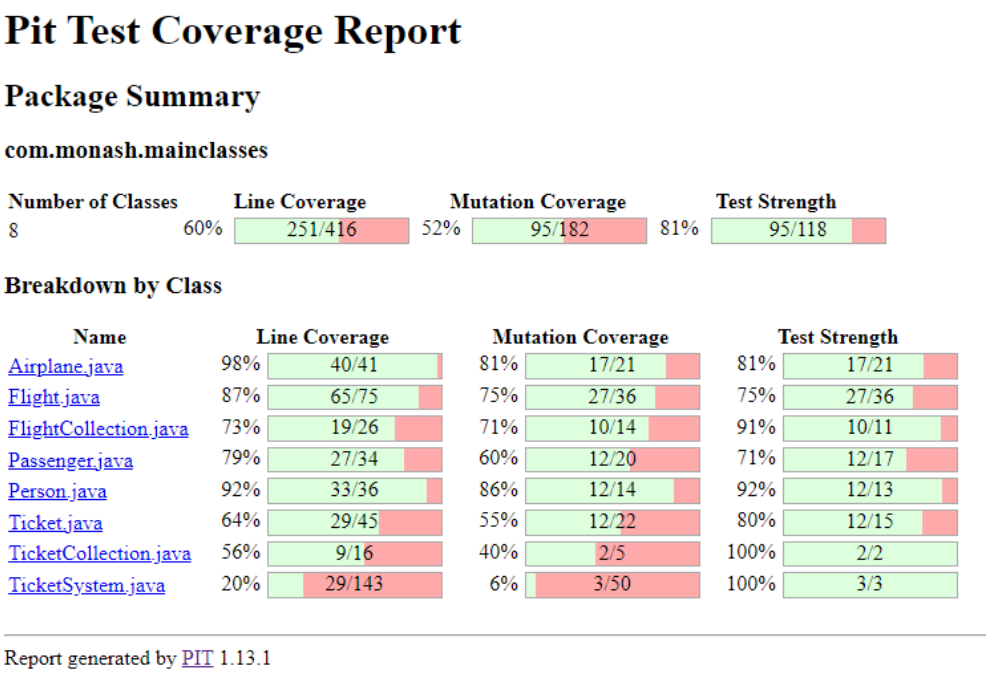
- Increasing Test Coverage: We scrutinized TicketSystem.java to identify areas not adequately covered by the tests. We added new test cases to cover these areas, ensuring no code paths were left untested.

- Reviewing Existing Test Cases: We revisited existing test cases for TicketSystem.java to identify why they weren't catching mutations effectively. We refined these tests to better detect potential faults.

- Introducing Edge Cases: We designed tests to handle edge cases, helping to improve the robustness of the suite and catch mutations that could lead to unexpected behavior.

- Continuous Mutation Testing: We decided to integrate mutation testing as part of our regular testing process to continuously identify areas of weakness and address them proactively.

By taking these steps, we sought to address the issues identified by mutation testing and strengthen our test suite.



Pit Test Coverage Report

Package Summary

com.monash.mainclasses

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	60%	251/416	52%	95/182	81%	95/118

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Airplane.java	98%	40/41	81%	17/21	81%	17/21
Flight.java	87%	65/75	75%	27/36	75%	27/36
FlightCollection.java	73%	19/26	71%	10/14	91%	10/11
Passenger.java	79%	27/34	60%	12/20	71%	12/17
Person.java	92%	33/36	86%	12/14	92%	12/13
Ticket.java	64%	29/45	55%	12/22	80%	12/15
TicketCollection.java	56%	9/16	40%	2/5	100%	2/2
TicketSystem.java	20%	29/143	6%	3/50	100%	3/3

Report generated by PIT 1.13.1

5. Conclusion

In conclusion, this report focused on the assessment and improvement of the provided codebase, utilizing SonarQube for quality analysis and PIT for mutation testing. Through the analysis, we identified areas for improvement in code structure, technical debt, code smells, security vulnerabilities, and test suite coverage. By addressing these issues, we made significant enhancements to the codebase and test suite. Refactoring, elimination of code smells, security implementations, and improved test coverage were key areas of improvement. This assignment has provided valuable insights into working with unfamiliar codebases, code quality assessment, and the importance of comprehensive testing. We are confident that the improvements made have enhanced the overall quality and reliability of the codebase.

6. Assessment

Task	Member
Code Quality Analysis, Mutation Testing	Akash Balakrishnan
Code merging, Mutation testing using PIT, SonarQube Setup.	Ashiklal Memanaparambil Asokalal
SonarQube setup and Changes to the code.	Jason Pius Dsouza

7. References

Pitest. (n.d.). Retrieved from http://pitest.org/

SonarQube. (n.d.). Documentation. Retrieved from https://docs.sonarqube.org/latest/

Pitest. (n.d.). Quickstart: Maven. Retrieved from https://pitest.org/quickstart/maven/