ASSIGNMENT 3 – SOFTWARE QUALITY MONITORING & IMPROVEMENT

FIT5171 - System Validation and Verification, Quality and Standards (S1 2024)

Lecturer: Yuan-Fang Li
Tutor: Saranya Alagarsamy
Class Day (Time): - Wednesday (12:00-14:00)
Submission date: 26th May 2024

Edward Fery – 29933226
Justin Wu – 33489468
Peichun Shih - 33475881

Table of Contents

Code Quality Analysis & Code Improvement

Code base	Test classes
Person Issues: - setAge – no validation for input less than zero - setGender – no validation for null input - setFirstName – no validation for null input - setSecondName – no validation for null input - Unnecessary using abstract class in Person Code improvement & integration: - Removed abstract class - Added the validation for all methods.	TestPerson Issues: - Inappropriate test case name used. - Test cases did not cover all methods. The original test cases only tested Passenger constructor with valid and invalid input and setGender with invalid input Test case improvement & integration: - Modified test case name - Added more test cases cover valid and invalid input for all methods
Passenger Issues: - Passenger constructor – no validation for phoneNumber (+615 and null value), passport, cardNumber and securityCode - setCardNumber – no validation (i.e. 16 digits and not null) - setSecurityCode – no validation (i.e. 3 digits and not null) - setPassport – no validation (i.e. alphanumeric and up to 9 chars long) - setPhoneNumber – incomplete validation (i.e. The original code did not consider +615 or null value) Code improvement & integration: - Added the validation for all methods.	PassengerTest Issues: - Inappropriate test case name used. - Test cases did not cover all methods. The original test cases only tested Passenger constructor with valid and invalid input, setPhone, setEmail and setPassport Test case improvement & integration: - Changed test case name - Added more test cases to ensure covering valid and invalid input for all methods
Airplane Issues: - Airplane constructor – no validation - setAirplaneID – no validation - setAirplaneModel – no validation - setBusinessSitsNumber – no validation - setEconomySitsNumber – no validation - setCrewSitsNumber – inappropriate validation (using System.out) - getAirPlaneInfo - No function was implemented in original codes. Code improvement & integration: - Added the validation for all methods. - Added List<Airplane> to store Airplane - Complete the getAirPlaneInfo method	AirplaneTest Issue: - Incorrect usage of setup() - Inappropriate test case name used. - Test cases did not cover all methods. The original test cases only tested seter and getter method with valid input for airplaneID, airplaneModel, businessSitsNumber, economySitsNumber and crewSitsNumber. Test case improvement & integration: - Changed test case name - Added more test cases to ensure covering valid and invalid input for all methods
Flight Issues: - Variable naming convention violated (example: flight_id) - No validation for setting departure date time to be before arrival date time, setAirplane, setFlightId, setDepartTo, setDepartFrom, setCode and setCompany. - Duplication – Date and Time validation Code improvement & integration: - Created methods to convert them to Timestamp and do validation - Created Date and Time validation method to avoid code duplication - Fixed naming convention (flight_id -> flightId) - Added universal null validation method	FlightTest Issues: - Access modifiers presence in all test methods - Test cases did not cover all methods and lines. Test case improvement & integration: - Removed all access modifiers in all test methods - Added and integrated new test methods from our own code base
FlightCollection Issues: - Flights arraylist access modifiers is public - No private constructor to hide the implicit public one (utility class) - Variable naming convention violated (example: flights_db) - Get and add flights method to accept arraylist Code improvement & integration: - Modified Flights arraylist modifiers to be protected - Created private constructor to hide the implicit public one - Fixed naming convention (flights_db -> flightsDb) - Modified Get and add flights method to accept List	FlightCollectionTest Issues: - Test method did not cover some test cases – getFlightInfo 1 city (invalid city name) and invalid flight id Test method did not testPrivateConstructor - Test case improvement & integration: - Added and integrated new test method from our code base to cover more test cases - Added testPrivateConstructor
Ticket Issues: - Ticket constructor – service tax is not applied to price and a price should always be applied to a ticket - setTicketId – no validation for ticket ID	TicketTest Issues: - A ticket object is created in every test case. - The original test cases did not test if the service tax is always applied to the price and if the ticket ID is valid.

- setPrice – no validation for price - saleByAge – no validation for age - serviceTax – should return price for further implementation - toString – should show complete information of a ticket Code improvement & integration: - Made service tax always apply to price when creating a ticket - Set default ticket price to $100 if no price is applied to a ticket. - Added validations - Modified serviceTax method to return an integer - Modified toString method to display complete ticket information	Test case improvement & integration: - Created a ticket object in setup(). - Added and integrated more test cases from our code base to cover valid and invalid input
TicketCollection Issues: - Tickets arraylist access modifiers is public - No private constructor to hide the implicit public one (utility class) - Variable naming convention violated (example: tickets_db) - Get and add tickets method to accept arraylist - addTickets – a complex method including various validations - getAllTickets – no parameters for city1 and city 2 and no return for the list of tickets - getTicketInfo – no validations for ticket ID Code improvement & integration: - Modified tickets arraylist modifiers to be protected - Created private constructor to hide the implicit public one - Fixed variable naming coneventions - Modified Get and add tickets method to accept List - Split addTickets method and created isTicketExisting and isTicketValid methods for ticket validations - Added parameters for getAllTickets and made it return a list of tickets. - Created displayAllTickets method to display tickets information - Added validations for ticket ID	TicketCollectionTest Issues: - The original test cases did not cover all valid and invalid ticket ID for testing methods. Test case improvement & integration: - Added and integrated test cases from our code base to test addTickets and getTicketInfo with valid and invalid ticket and ticketID respectively.
TicketSystem The TicketSystem code remains largely unmodified from the original version, leading to integration issues. Issues (although there are plethora of issues, here are some): - return; is unnecessary for void methods - buyTicket – accept null passenger details - variable naming conventions (ticket_id) Code improvement & integration: - Integrated and replaced with our team code base's methods and added missing methods such as validations and logic to reduced cognitive complexity.	TicketSystemTest Issues: - Test cases did not cover all methods (example showTicket) - Test cases did not simulate choosing and buying ticket output flow and logic Test case improvement & integration: - Integrated with our team code base's test methods to simulate possible scenarios for ticket choosing and buying logic. - Simulated possible user inputs - Checked system output - Implemented appropriate @BeforeEach and @AfterEach initiatives

JUnit – Test Coverage

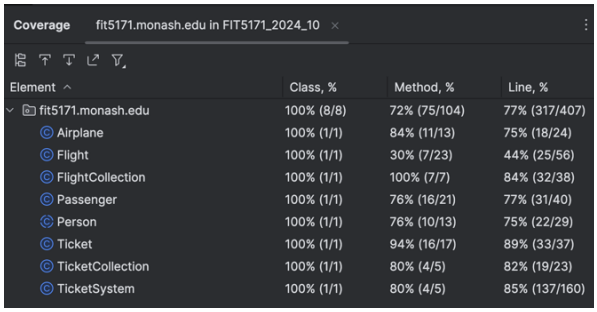


Figure 1.1 JUnit Test Coverage Before Integration & Improvement

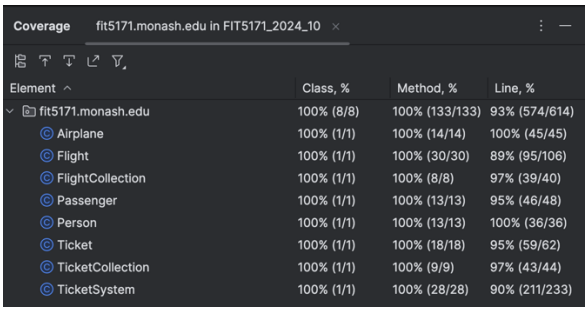


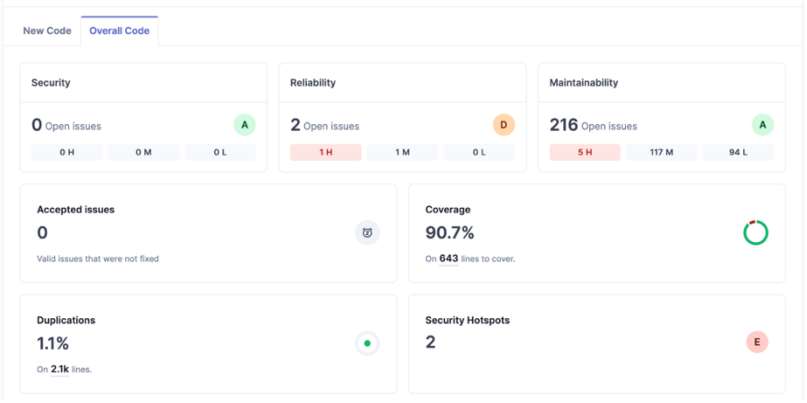
Figure 1.2 JUnit Test Coverage After Integration & Improvement

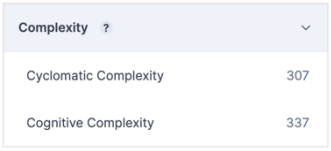
The JUnit test coverage before integration and improvement showed significant gaps in method and line coverage among various classes. As shown in Figure 1.1, while class coverage was at 100% for all classes, method coverage varied, with some classes like Flight having only 30% method coverage and FlightCollection with 100%. Line coverage also varied, with the TicketSystem class having 85%, and Flight only 44%.

After the integration and improvement efforts, a notable improvement is in test coverage. Figure 1.2 highlights these improvements, with method coverage for all classes reaching 100% and line coverage largely increasing, achieving 93% overall. Specific classes such as Airplane and Person have their line coverage rise to 100%, indicating thorough testing and better code reliability. Additionally, in Flight class, a huge enhancement was made to increase the method coverage from 30% to 100% and the line coverage from 44% to 89% respectively. This comprehensive testing approach ensures that the codebase is more robust and less prone to defects, offering a foundation for future development and maintenance.

SonarQube – Code Quality Analysis and Monitoring

Figure 2.1 Initial SonarQube code analysis summary

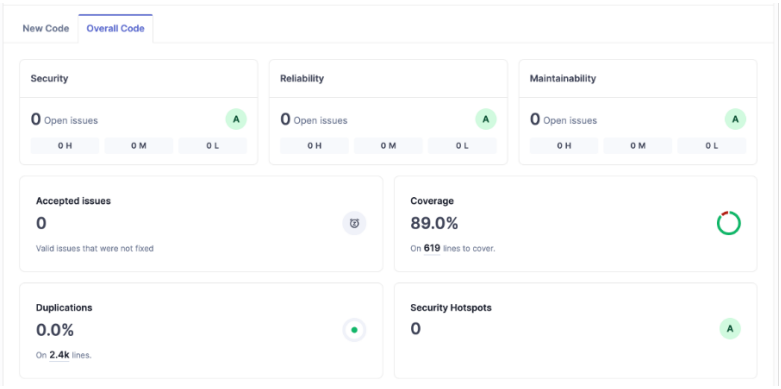


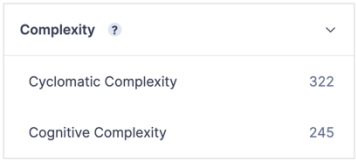


The initial analysis indicated several key metrics that need attention. The security metric showed no open issues, which is positive. However, there are two open issues under reliability, with one classified as high severity and one as medium. The maintainability metric shows 216 open issues, with five classified as high severity, 117 as medium, and 94 as low that need to be addressed.

The overall test coverage stands at 90.7% across 643 lines of code, which is commendable. However, the complexity metrics, with a cyclomatic complexity of 307 and cognitive complexity of 337, suggest that there is significant room for improvement in simplifying and optimizing the code. Additionally, there are 1.1% duplications across 2.1k lines of code and two security hotspots that need to be addressed.

Figure 2.2 SonarQube code analysis summary after improvements

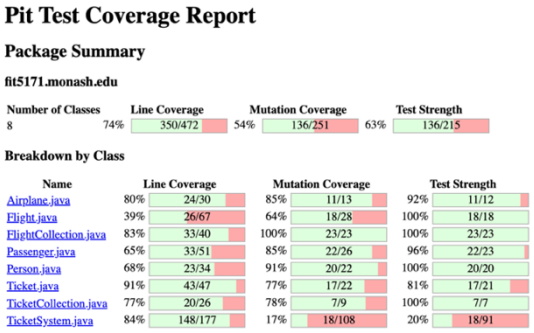




Following code improvements and optimizations, a subsequent analysis showed considerable enhancements. Security, reliability, and maintainability metrics now report zero open issues. This improvement signifies that our code is more robust and easier to maintain. The overall test coverage slightly decreased to 89.0% across 619 lines of code due to code refactoring.

The complexity metrics also improved, with cyclomatic complexity reducing to 322 and cognitive complexity to 245. This reduction indicates that our codebase is now simpler and more understandable, reducing the risk of errors and making future maintenance easier. Furthermore, we achieved zero duplications across 2.4k lines of code and no security hotspots, highlighting the effectiveness of our refactoring efforts.

PIT Test – Mutation Testing



Pit Test Coverage Report

Package Summary

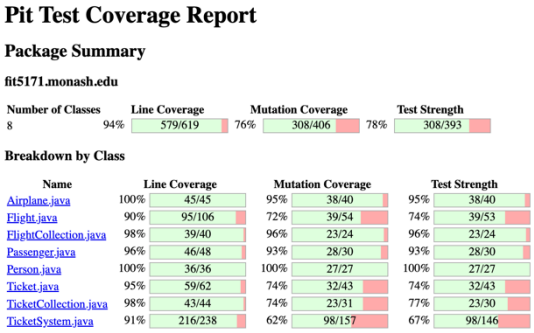
fit5171.monash.edu

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	74%	350/472	54%	136/251	63%	136/215

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Airplane.java	80%	24/30	85%	11/13	92%	11/12
Flight.java	39%	26/67	64%	18/28	100%	18/18
FlightCollection.java	83%	33/40	100%	23/23	100%	23/23
Passenger.java	65%	33/51	85%	22/26	96%	22/23
Person.java	68%	23/34	91%	20/22	100%	20/20
Ticket.java	91%	43/47	77%	17/22	81%	17/21
TicketCollection.java	77%	20/26	78%	7/9	100%	7/7
TicketSystem.java	84%	148/177	17%	18/108	20%	18/91

Figure 3.1 PIT Test Before Integration & Improvement



Pit Test Coverage Report

Package Summary

fit5171.monash.edu

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	94%	579/619	76%	308/406	78%	308/393

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Airplane.java	100%	45/45	95%	38/40	95%	38/40
Flight.java	90%	95/106	72%	39/54	74%	39/53
FlightCollection.java	98%	39/40	96%	23/24	96%	23/24
Passenger.java	96%	46/48	93%	28/30	93%	28/30
Person.java	100%	36/36	100%	27/27	100%	27/27
Ticket.java	95%	59/62	74%	32/43	74%	32/43
TicketCollection.java	98%	43/44	74%	23/31	77%	23/30
TicketSystem.java	91%	216/238	62%	98/157	67%	98/146

Figure 3.2 PIT Test After Integration & Improvement

The PIT Mutation Testing results demonstrated significant improvements after integration. Initially, the package summary indicated 74% line coverage, 54% mutation coverage, and 63% test strength. Airplane.java had the highest line coverage (80%), while TicketSystem.java had the lowest mutation coverage (17%).

After integration, line coverage increased to 94%, mutation coverage to 76%, and test strength to 78%. Notable improvements included Airplane.java and Person.java achieving 100% line coverage, with Person.java also achieving 100% mutation coverage. TicketSystem.java saw line coverage rise to 91%, mutation coverage to 62%, and test strength to 67%. Overall, the integration significantly enhanced code quality and test effectiveness, with line coverage improving from 74% to 94%, mutation coverage from 54% to 76%, and test strength from 63% to 78%.

Implementing CI/CD

The implementation of the CI pipeline involved several key components, primarily focusing on SonarQube for continuous code quality analysis and using GitLab CI for managing the continuous integration process. To achieve this, we configured:

1. SonarQube CI/CD integration
   We configured SonarQube for local development using Docker on port 9000. This setup allowed us to run SonarQube locally, providing a platform for continuous code quality analysis. By running SonarQube locally, we could ensure that the code met quality standards before pushing changes to the shared repository.
2. Pom.xml configuration
   The pom.xml file was updated to include the necessary plugins and dependencies for SonarQube analysis. Specifically, the JaCoCo plugin was added for coverage report generation for SonarQube. Additionally, the SonarQube plugin configuration was integrated into the Maven build lifecycle. This configuration enabled the automatic execution of SonarQube report generation.
3. Gitlab CI configuration
   The gitlab-ci.yml file was configured with three stages: build, sonarqube-check, and sonarqube-vulnerability-report. In the build stage, the project was compiled and tested. The sonarqube-check stage involved running SonarQube analysis to check the code quality. Finally, the sonarqube-vulnerability-report stage generated a detailed report on any vulnerabilities found. Additionally, environment variables such as SONAR_TOKEN and SONAR_HOST_URL were configured to authenticate and connect to the SonarQube server. A GitLab runner was also created to execute the CI/CD pipeline jobs.

References

OpenAI. (2023). *ChatGPT* (April 2023 version) [Large language model] http://chat.openai.com/chat

---

I acknowledge the use of ChatGPT (https://chat.openai.com/) to generate materials for background research and self-study in doing this assessment. I entered the following prompts:

1. SonarQube coverage showing 0%, how to fix this?
2. How to integrate JaCoCo to the pom.xml
3. JaCoCo report is not generated due to missing execution file
4. JaCoCo error <error prompt>
5. PITest plugin does not work
6. Mavesurefire argline does not work for JaCoCo
7. How to configure runner for gitlab
8. Runner for macos
9. How to get public ipaddress for localhost (SonarQube)
10. Help me improve this section <paragraph>

The output from the generative artificial intelligence was used to:

1. Gather insights JaCoCo and SonarQube integration
2. Gathe insights to properly integrate PIT Test plugin
3. Improve writings.

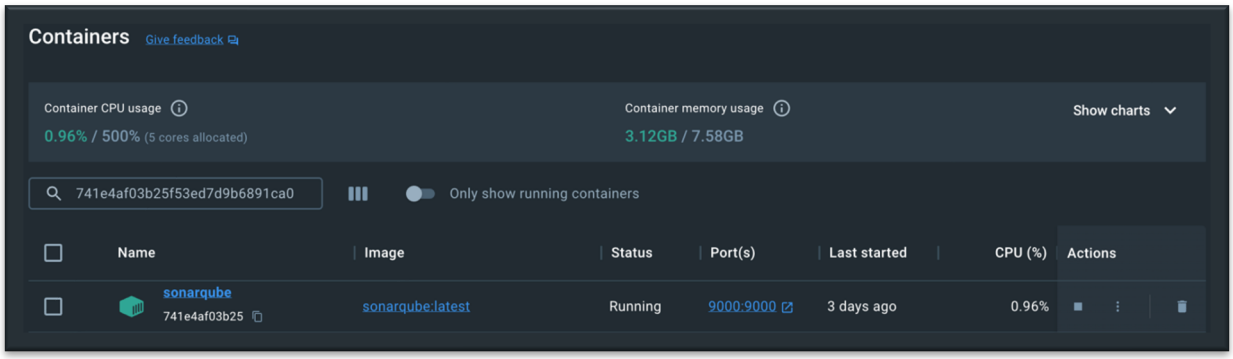
Appendices

Git URL

Git URL: https://git.infotech.monash.edu/fit5171_21/a3.git

SonarQube Integration

Docker



pom.xml

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <junit.jupiter.version>5.10.2</junit.jupiter.version>
    <mockito.version>4.5.1</mockito.version>
    <sonar.projectKey>fit5171_21_a3_257cdc33-a8ba-46de-817d-49888ac2a2b4</sonar.projectKey>
    <sonar.projectName>a3</sonar.projectName>
    <sonar.qualitygate.wait>true</sonar.qualitygate.wait>
    <jacoco.version>0.8.12</jacoco.version>
    <sonar.coverage.jacoco.xmlReportPaths>target/site/jacoco/jacoco.xml</sonar.coverage.jacoco.xmlReportPaths>
    <jacoco.agent.path>${project.basedir}/lib/org.jacoco.agent-${jacoco.version}-runtime.jar</jacoco.agent.path>
    <jacoco.argLine>-javaagent:${jacoco.agent.path}=destfile=${project.build.directory}/jacoco.exec</jacoco.argLine>
    <sonar.exclusions>src/main/java/fit5171/monash/edu/Main.java</sonar.exclusions>
</properties>
```

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>${jacoco.version}</version>
    <executions>
        <execution>
            <id>prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
            <configuration>
                <propertyName>jacoco.agent.argLine</propertyName>
            </configuration>
            <phase>initialize</phase>
        </execution>
        <execution>
            <id>report</id>
            <phase>verify</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```
<plugin>
    <groupId>org.sonarsource.scanner.maven</groupId>
    <artifactId>sonar-maven-plugin</artifactId>
    <version>3.9.1.2184</version>
</plugin>
```

Integrated JaCoCo for coverage report generation and ensure SonarQube can display coverage %.

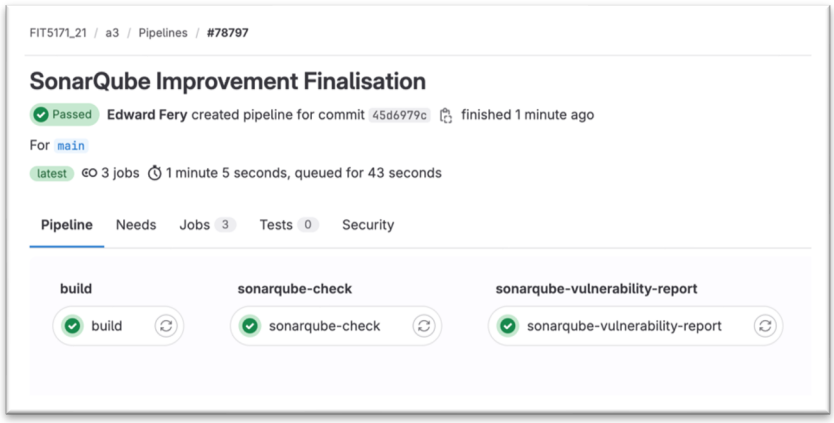
(Note: we encountered significant challenges in integrating JaCoCo plugins to our dependency configuration, mainly due to version incompatible and maven ability to download the appropriate JaCoCo jar execution file. Hence we manually download the JaCoCo file and instruct maven to use that jar file.)

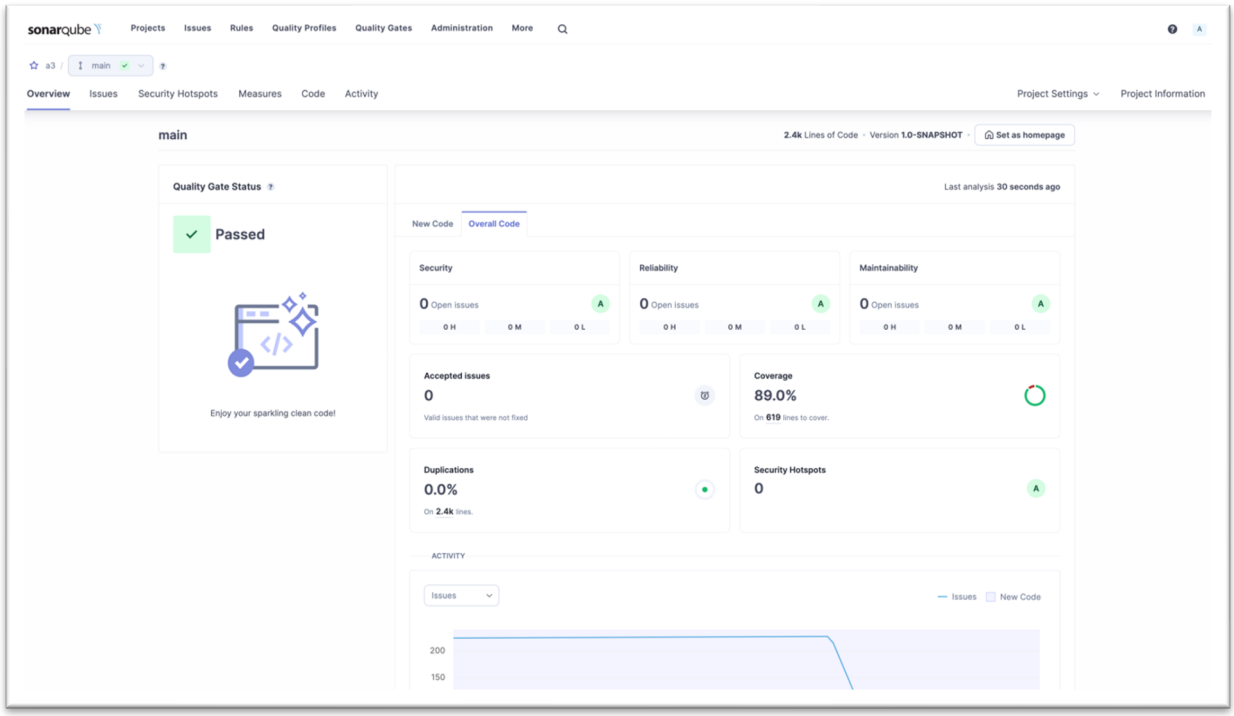
.gitlab-ci.yml

Configured tags to use ‘macos’ runner and integrate JaCoCo for SonarQube coverage report generation.

```
stages:
  - build
  - sonarqube-check
  - sonarqube-vulnerability-report

build:
  stage: build
  image: maven:3-eclipse-temurin-17
  tags:
    - macos
  script:
    - mvn clean install

sonarqube-check:
  stage: sonarqube-check
  image: maven:3-eclipse-temurin-17
  tags:
    - macos
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"  # Defines the location of the
analysis task cache
    GIT_DEPTH: "0"  # Tells git to fetch all the branches of the project, required
by the analysis task
  cache:
    key: "${CI_JOB_NAME}"
    paths:
      - .sonar/cache
  script:
    - mvn clean verify jacoco:report sonar:sonar
  artifacts:
    paths:
      - target/site/jacoco/jacoco.xml
  allow_failure: true
  only:
    - merge_requests
    - master
    - main
    - develop

sonarqube-vulnerability-report:
  stage: sonarqube-vulnerability-report
  tags:
    - macos
  script:
    - 'curl -u "${SONAR_TOKEN}:"
"${SONAR_HOST_URL}/api/issues/gitlab_sast_export?projectKey=fit5171_21_a3_257cdc33-
a8ba-46de-817d-
49888ac2a2b4&branch=${CI_COMMIT_BRANCH}&pullRequest=${CI_MERGE_REQUEST_IID}" -o gl-
sast-sonar-report.json'
  allow_failure: true
  only:
    - merge_requests
    - master
    - main
    - develop
  artifacts:
    expire_in: 1 day
    reports:
      sast: gl-sast-sonar-report.json
  dependencies:
    - sonarqube-check
```

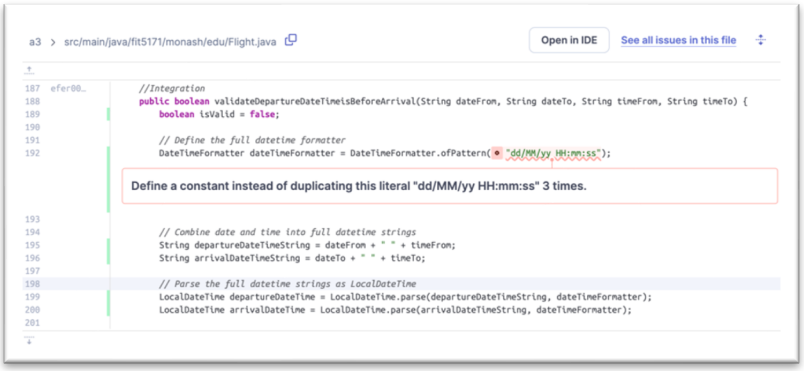
GitLab Pipeline and SonarQube Dashboard





SonarQube Issues – examples

Some critical level of severity includes:

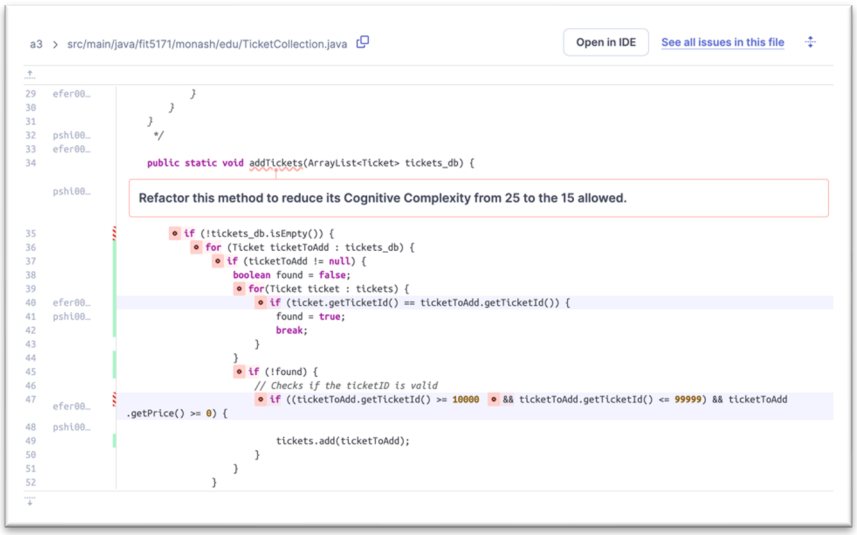


1. Flight.java

Define a constant instead of duplicating this literal "dd/MM/yy HH:mm:ss" 3 times.

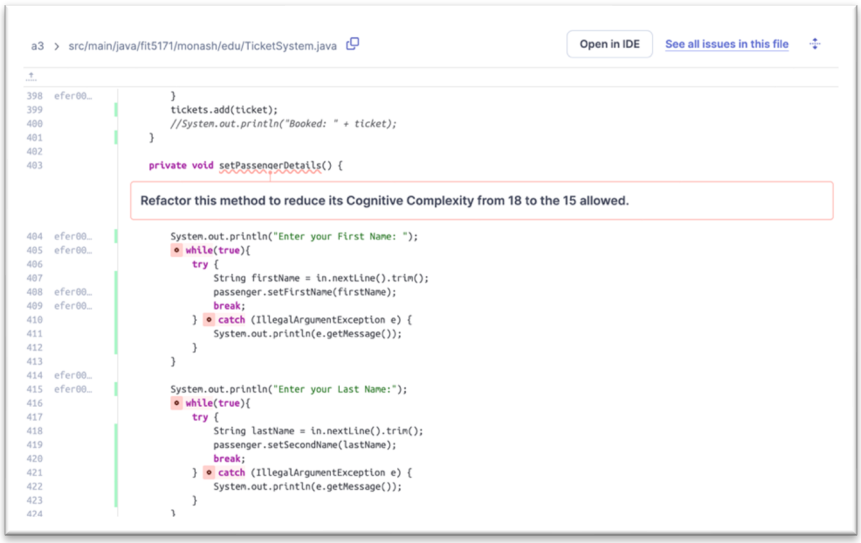
2. TicketCollection.java

Refactor addTickets method to reduce its Cognitive Complexity from 25 to the 15 allowed.



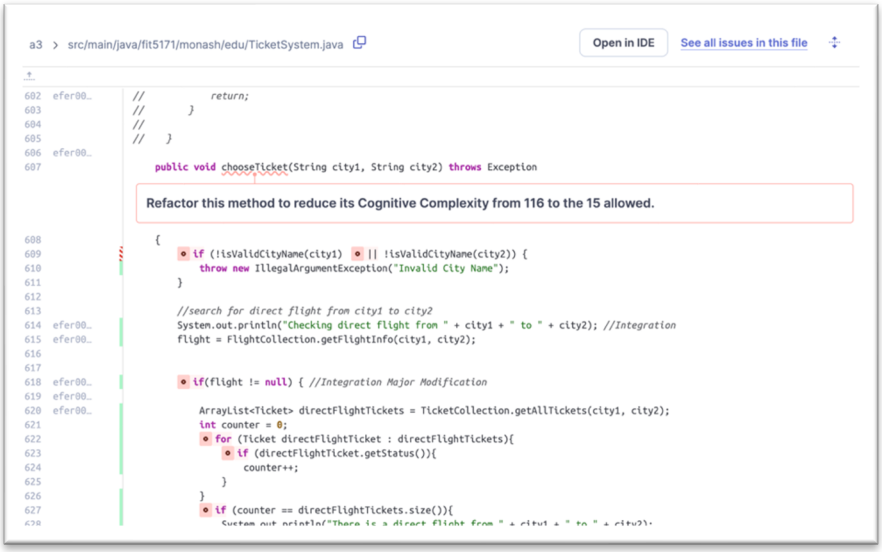
3. TicketSystem.java

Refactor setPassengerDetails method to reduce its Cognitive Complexity from 18 to the 15 allowed.



4. TicketSystem.java

Refactor chooseTicket method to reduce its Cognitive Complexity from 116 to the 15 allowed.



PIT Test Integration

pom.xml

```xml
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.16.1</version>
    <dependencies>
        <dependency>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-junit5-plugin</artifactId>
            <version>LATEST</version>
        </dependency>
    </dependencies>
    <configuration>
        <targetClasses>
            <param>fit5171.monash.edu.*</param>
        </targetClasses>
        <targetTests>
            <param>fit5171.monash.edu.*Test</param>
        </targetTests>
        <excludedClasses>
            <param>fit5171.monash.edu.Main</param> <!-- Exclude Main.java -->
        </excludedClasses>
        <outputFormats>
            <param>XML</param>
            <param>HTML</param>
        </outputFormats>
    </configuration>
</plugin>
```

Pit Test Coverage Report

Package Summary

fit5171.monash.edu

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
8	95%	611/643	75%	294/391	77%	294/381

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Airplane.java	100%	45/45	95%	38/40	95%	38/40
Flight.java	94%	94/100	74%	39/53	74%	39/53
FlightCollection.java	95%	37/39	96%	23/24	96%	23/24
Passenger.java	97%	58/60	95%	36/38	95%	36/38
Person.java	100%	36/36	100%	27/27	100%	27/27
Ticket.java	96%	65/68	74%	32/43	74%	32/43
TicketCollection.java	98%	44/45	70%	19/27	73%	19/26
TicketSystem.java	93%	232/250	58%	80/139	62%	80/130