Module 7 Project Two

Kemal Cankurt

SNHU

Course Number: CS-320

Dr. Kraya

Summary and Reflections Report

I implemented a unit testing strategy for the Contact, Task, and Appointment services. This helped me ensure that the services delivered were top-notch and reliable. The thorough testing process helped identify and fix any possible issues before they affected the end users. Such an approach will also ensure that users have a seamless experience in real-world examples. Also, prioritizing comprehensive validation and early issue detection will help meet users' expectations. Overall, I delivered services that met the standards of quality and reliability.

My testing of the contact service was careful, ensuring its accuracy and reliability. The service's high quality and consistency were maintained through tests that verified the uniqueness of contact IDs, enforced constraint validations, and checked specific criteria such as the phone number length. These tests were evidence of the service's commitment to delivering a robust and dependable contact management system. Below is an example:

*@Test*

public void Given\_Contact\_When\_FirstName\_Exceeds\_10\_Char\_Limit\_Then\_Throw\_IllegalArgumentException() {

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Contact("123", "KemalKemal123", "Cankurt", "1234567890", "2500 N River Rd, Manchester, NH 03106");

});

}

Within the task service, a key focus was placed on validating the task ID's uniqueness and verifying the name and description fields' accuracy per the specified requirements. An illustration of this validation involved ensuring that the task description did not exceed 50 characters. By carefully conducting these checks, the task service aimed to guarantee adherence to standards and precision in the task-related data. Below is another unit test to make sure the length of the description field does not exceed 50 characters:

*@Test*

void Given\_Task\_When\_Description\_Length\_Is\_More\_Than\_50\_Then\_Throw\_IllegalArgumentException() {

Exception exception = Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Task("1", "Task Grocery", "Apples Bananas Strawberries Avocados Bell Peppers Carrots Broccoli Garlic Lemons/Limes");

});

Assertions.*assertTrue*(exception.getMessage().contains("Please enter a valid description"));

}

Ensuring the accuracy and relevance of the appointment scheduling system was a priority. A key measure was to validate that the appointment date is in the future, not the past. This validation was crucial as it prevented outdated or invalid appointments from being processed, which could lead to confusion and errors. By checking the appointment date, the system could guarantee that appointments were scheduled only for valid and relevant dates, thereby enhancing the system's overall efficiency and reliability. Below is a test case which illustrates of validating the appointment date:

*@Test*

void Given\_Appointment\_When\_Date\_Is\_In\_Past\_Then\_Throw\_Error() {

Date now = new Date();

Calendar calendar = Calendar.*getInstance*();

calendar.setTime(now);

calendar.add(Calendar.***DATE***, -1);

Date yesterday = calendar.getTime();

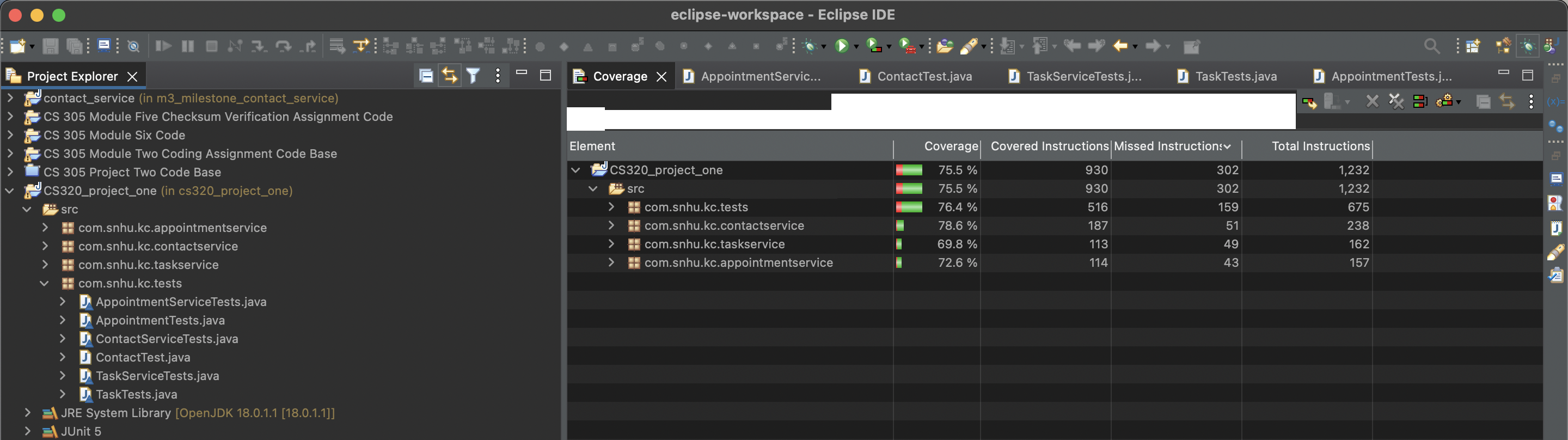
*assertThrows*(IllegalArgumentException.class, () -> {

new Appointment("1", yesterday, "Check-up");

});

}

By utilizing JUnit tests, I identified and addressed potential issues early in the development process, saving time and resources that I would have otherwise spent on debugging and fixing errors later. The comprehensive test coverage of around 80% has provided a safety net for the system, allowing for continuous integration and deployment with confidence that new changes will not disrupt existing functionality. Below is a screenshot of testing coverage results from Eclipse:



As I wrote JUnit tests, I better understood the application's functional dynamics. This allowed me to identify and resolve potential issues before they affected the user experience. I tested each requirement thoroughly and ensured the system was stable and error-free. Additionally, I created robust error-handling mechanisms that would prevent the application from crashing or causing data loss. Overall, this experience helped me to develop a more comprehensive understanding of software testing and its importance in delivering high-quality applications.

The project heavily relied on unit testing, a critical method that involves testing small code units to ensure they behave as expected. This approach is essential for catching bugs early on and verifying that each module can function independently. Despite the emphasis on unit testing, performance testing was overlooked in this project. Performance testing evaluates how well the application performs under various loads, which is crucial for understanding response times and scalability when faced with high user traffic. Incorporating performance testing into the project could have provided valuable insights into the system's capabilities under stress. By simulating heavy user loads, developers can identify potential bottlenecks and optimize the application for extreme performance. While unit testing is vital for catching bugs at the code level, performance testing offers a broader perspective on the application's overall efficiency and reliability. In future projects, balancing unit and performance testing can lead to a more robust and high-performing software system.

Furthermore, disciplined software engineering and quality practices play a crucial role in the early detection and resolution of potential issues. Through comprehensive testing and validation at each development stage, problems are identified and rectified before they escalate. Ultimately, these practices are indispensable for ensuring the dependability and resilience of any software system, offering assurance that the software being tested remains efficient and reliable even when facing high usage levels.

Developers or software engineering teams must avoid technical debt to keep the software working efficiently and maintain quality. Technical debt can have severe consequences that can harm the software's quality and efficiency. Hence, it's essential to prioritize avoiding technical debt and always follow the best practices to keep the software running smoothly. To achieve long-term success, developers should always maintain this aspect. Furthermore, to reduce the amount of technical debt, having at least two other developers review any code before it's merged is a good idea. This review should check for code quality, adherence to project standards, and potential long-term impacts that might introduce technical debt. In addition, developers need to be well-trained and keep up with the latest best practices. There are many resources on platforms like Pluralsight or Udemy to help teams achieve this goal.

References

Fowler, M. (2019, May 21). Technical debt quadrant. <https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>

GeeksforGeeks. Performance testing (Software Testing). <https://www.geeksforgeeks.org/performance-testing-software-testing>

Atlassian. Code coverage.

<https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>