Summary and Reflections Report CS 320

In this project, unit testing was employed to verify the functionality of the contact, task, and appointment services. The testing approach focused on ensuring compliance with the given requirements, validating input constraints, and checking system behavior under edge cases. Each feature underwent systematic unit testing using JUnit to verify the correctness of object creation, modification, and deletion functions.

The testing approach was strongly aligned with the software requirements. For instance, the ContactService required a unique ID with a maximum length of 10 characters, mandatory fields for first name, last name, phone number, and address. The unit tests validated these constraints by explicitly testing null values, invalid lengths, and format violations. One such test confirmed that an exception was thrown when attempting to create contact with an overly long ID:

assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", "John", "Doe", "1234567890", "123 Elm Street"));

The overall quality of the JUnit tests was ensured through high test coverage, targeting both expected and edge-case scenarios. Coverage metrics demonstrated that more than 80% of the application code was exercised, validating the robustness of the tests. By testing expected values and negative cases, the JUnit tests effectively verified system stability.

To ensure the code was technically sound, structured assertions were used to confirm that system behavior matched expectations. In TaskServiceTest, validations were implemented to confirm that task updates only affected modifiable fields while keeping immutable fields unchanged:

service.updateTask("task123", "Updated Title", "Updated Description");

assertEquals("Updated Title", task.getTitle());

assertEquals("Updated Description", task.getDescription());

To optimize efficiency, test cases avoided redundant checks and were structured to minimize execution overhead. For example, loops in AppointmentService were designed to return early once the desired appointment was located, reducing unnecessary iterations and improving performance.

This project leveraged unit testing and boundary value analysis as primary verification methods. Unit testing provided granular validation for each method, ensuring functions operated correctly in isolation. Boundary value analysis tested values at their limits (e.g., verifying a contact name with exactly 10 characters), ensuring compliance with system constraints.

Other testing techniques, such as integration testing and exploratory testing, were not used in this project. Integration testing evaluates interactions between components, ensuring seamless communication, which would be applicable if this project involved database interactions or API calls. Exploratory testing, which relies on manual interaction to uncover unexpected behavior, was not applied as the project focused on predefined unit tests. However, these techniques could be valuable in large-scale applications where system behavior is influenced by external dependencies.

While working on this project, a cautious and detail-oriented mindset was adopted to ensure software correctness. Given the interdependencies of various functions, recognizing how one function could affect another was crucial. For example, failing to validate appointment dates could lead to scheduling conflicts or logical inconsistencies.

To limit bias in testing, assertions were implemented based on predefined requirements rather than personal expectations of how the software should behave. Additionally, systematic negative testing was performed to confirm that errors were appropriately handled. Bias in testing one’s own code is a concern, as developers may unconsciously assume correctness. To mitigate this, conducting peer reviews or leveraging automated test coverage reports can be helpful.

A commitment to software quality was maintained throughout the project, reinforcing the importance of discipline in engineering practices. Cutting corners in software testing can lead to severe consequences, such as security vulnerabilities or system failures. To avoid technical debt, best practices like modular development, reusable components, and consistent documentation will be adhered to in future projects. Incorporating continuous integration with automated test execution will also ensure code quality is maintained over time.

This project reinforced the importance of thorough unit testing, systematic validation, and quality assurance in software development. By aligning tests with requirements, maintaining high code coverage, and applying structured debugging strategies, the robustness of the application was ensured. The experience gained in this project will serve as a foundation for applying more advanced testing methodologies in larger-scale software systems.