Tiffany Thai

Southern New Hampshire University

CS 320: Software Testing, Automation, and Quality Assurance

Dr. Karl Lewis

December 15, 2024

This report provides an overview of the software testing strategies employed during the development of the contact, task, and appointment services for a mobile application. It outlines the approach used to ensure alignment with software requirements, evaluates the effectives and efficiency of the Junit tests, and reflects on the testing techniques and mindset adopted throughout the project.

The unit testing approach for the mobile application’s contact, task, and appointment services demonstrated a comprehensive strategy that closely aligned with software requirements. Each service was subjected to rigorous testing that covered critical functional scenarios:

* Contact Service: Tests validated the creation, updating, and deletion of contact entries, focusing on handling null inputs, ensuring unique IDs, and validating constraints like name and phone number lengths.
* Task Service: Tests ensured that tasks could added, retrieved, and deleted correctly. Specific scenarios included verifying task descriptions lengths and ensuring non-null task IDs.
* Appointment Service: Tests confirmed proper scheduling, retrieval, and deletion of appointments. Scenarios included verifying valid dates and preventing duplicate entries.

The unit tests written ensured that each service functioned as expected and met its specified requirements. The Junit tests demonstrated high-quality coverage by addressing multiple scenarios: successful operations, error handling, edge cases, and boundary conditions. Key evidence of test effectiveness include the use of assertThrows() to validate error conditions, consistent use of unique identifiers, and thorough validation of object states after operations. The code coverage analysis demonstrated a near 100% coverage, confirming that critical paths were tested.

The tests exhibited technical soundness through:

* Clear, focused test method names describing their purpose
* Consistent use of setup methods with @BeforeEach
* Proper exception handling
* Precise assertions

Examples from ContactServiceTest.java:

1. @Test void testUpdateContact() {

2. Contact contact = new Contact("1234567890", "John", "Doe", "1234567890", "123 Main St");

3. contactService.addContact(contact);

4. contactService.updateContact("1234567890", "Jane", "Smith", "0987654321", "456 Elm St");

5.

6. assertEquals("Jane", contactService.getContact("1234567890").getFirstName());

7. assertEquals("Smith", contactService.getContact("1234567890").getLastName());

8. }

Efficiency was achieved through minimalist test methods, reuse of setup methods, avoiding redundant test scenarios, and quick and focused assertions.

The primary testing techniques employed in this project included unit testing, boundary value testing, and error/exception testing. Unit testing allowed for isolated verification of individual components, ensuring that each method and class functioned as expected. Boundary value testing focused on examining the edges of input ranges, helping identify potential issues at input limits. Error and exception testing validated the application's ability to handle unexpected scenarios gracefully.

While these techniques provided solid coverage, other potential approaches like integration testing and performance testing were not implemented. Integration testing would have explored interactions between different components, while performance testing could have assessed the system's efficiency under various conditions.

Throughout this project, I approached software testing with a methodical and cautious mindset, recognizing that thorough testing is crucial for developing reliable software. My approach went beyond surface-level verification, focusing on identifying potential vulnerabilities and edge cases that could compromise the system's integrity.

The complexity of software development requires a deep understanding of how different components interact. In practice, this meant carefully designing tests that examined not just the happy path, but also potential failure scenarios. For instance, in the appointment service, I implemented tests to ensure:

* Appointments cannot be created with past dates
* Duplicate appointment IDs are prevented
* Appropriate error handling occurs when attempting to delete non-existent appointments

These tests demonstrate the importance of comprehensively understanding the interrelationships between different parts of the system.

Mitigating bias is a critical challenge in software development, particularly when testing one's own code. To address this, I used an objective approach focused strictly on meeting specified requirements and functionality. The goal was to evaluate the code based on its performance and reliability, rather than preconceived notions.

In a professional context, bias can be particularly problematic. Developers naturally tend to be overly optimistic about their own code, which can lead to overlooking potential issues. To counteract this, best practices include:

* Conducting peer reviews (Team, 2024)
* Implementing comprehensive automated testing
* Approaching code evaluation with a critical and impartial mindset

My testing strategy deliberately attempted to identify potential failure points, such as adding duplicate contacts, updating non-existent tasks, and handling boundary conditions. This approach ensures a more rigorous and objective evaluation of the code's capabilities.

Maintaining discipline in software development is fundamental to creating high-quality, maintainable code. Technical debt represents the long-term consequences of taking shortcuts or implementing suboptimal solutions. Just as financial debt accumulates interest, technical debt becomes increasingly costly and challenging to address over time.

To mitigate technical debt and maintain high-quality standards, I am committed to:

* Writing clear, concise, and well-documented code (What Is Technical Debt: Common Causes & How to Reduce It | DigitalOcean, n.d.)
* Implementing comprehensive unit testing
* Regularly reviewing and refactoring existing code
* Staying current with industry best practices and emerging technologies

A concrete example from this project is the approach to error handling. Rather than simply ensuring positive scenarios work correctly, I developed tests that verified the system's behavior under various error conditions. In the task service, for instance, attempting to update a non-existent task triggers an error message.

The unit testing approach developed for the mobile application's contact, task, and appointment services demonstrates a comprehensive and methodical strategy for ensuring software reliability. By implementing thorough JUnit tests, I successfully validated the functionality of each service. I addressed specified requirements and potential error scenarios. This testing process not only confirmed the application’s functionalities but also provided confidence in the application's ability to meet the customer's specific needs.

**References**

Team, K. (2024, September 17). Cognitive Biases in Software Testing: A Guide to Overcome. *Katalon*. https://katalon.com/resources-center/blog/cognitive-biases-in-software-testing#:~:text=Peer%20Review%3A%20having%20multiple%20testers,experiences%20of%20a%20single%20tester.

*What is technical debt: Common causes & how to reduce it | DigitalOcean*. (n.d.). https://www.digitalocean.com/resources/articles/what-is-technical-debt