Austin Sarkis

12/11/2024

CS 320

**Project Two: Summary and Reflection**

**Testing Approach**

When developing and testing each of the services, I prioritized ensuring my unit tests were closely aligned with the requirements provided. I aimed to validate every functionality outlined in the specifications, which I accomplished by designing tests that mapped directly to those requirements. For example, the Contact class required the contactId to be unique, non-null, and unmodifiable. To test this, I wrote a test in the ContactTest file like

A computer screen shot of text

Description automatically generated

to confirm that null values were rejected correctly. Similarly, the Task class had constraints on taskId length and uniqueness. I tested this in TaskTest using



By directly testing the required aspect constraint, I was confident that my tests reflected the functionality expected by the requirements.

**Quality of JUnit Tests and Coverage**

I paid close attention to the quality of my JUnit tests by ensuring high coverage of all possible scenarios and all lines of code were tested. I tested valid cases, edge cases, and error conditions extensively. For instance, in ContactServiceTest, I made sure the system could handle duplicate contacts by writing a test like

A screen shot of a computer code

Description automatically generated

This allowed me to verify that the service could effectively identify and prevent duplicates. The tests didn't just cover happy paths; they were comprehensive enough to simulate and catch potential misuses of the code, which is essential for reliable software.

**Ensuring Technical Soundness**

To ensure my code was technically sound, I focused on testing specific behaviors and constraints for each class. For example, in TaskTest, I wrote a test to check that descriptions exceeding the maximum allowed length would throw an exception:



Similarly, in TaskServiceTest, I validated the behavior of the delete functionality by ensuring that after a task was removed, it could no longer be retrieved. This was done with

A computer screen shot of text

Description automatically generated

Writing tests like these gave me confidence that my code adhered to the requirements and behaved as intended in all scenarios.

**Efficient Code**

Efficiency in my tests was something I intentionally designed for by using tools like grouped assertions (assertAll) to test multiple conditions in a single test method. For instance, in TaskTest, I used assertAll to validate multiple fields at once:

A computer screen shot of text

Description automatically generated

With this approach, instead of creating individual tests to evaluate each aspect of the same object, we can combine them under one overarching test parameter, reducing the number of different tests.

**Software Testing Techniques Employed**

The primary technique I employed was unit testing, which focuses on testing individual code components in isolation. For example, in the Contact class, I created tests to ensure that fields such as contactId, firstName, and lastName met the constraints outlined in the requirements. In ContactTest, I used assertions like:

A computer screen with blue and green text

Description automatically generated

This test confirmed that invalid inputs led to exceptions, demonstrating that the class effectively managed these scenarios. Unit testing was essential for verifying the functionality of each class, including field validation, ensuring proper updates, and handling service operations such as adding and deleting objects.

Another technique I employed was boundary testing, which involves checking the limits of acceptable input. For example, in the TaskTest file, I ensured that string lengths for taskId, name, and description did not exceed their respective maximums. In TaskServiceTest, I verified that attempting to add tasks with duplicate IDs triggered appropriate errors:

A computer screen shot of text

Description automatically generated

Boundary testing helped confirm that the system functioned correctly at the limits of acceptable inputs, such as the maximum allowed string lengths and valid dates.

I also used exception testing extensively. This technique ensured that the system handled invalid inputs gracefully by throwing appropriate exceptions. For example, in AppointmentTest, I validated that dates in the past or null were rejected:

A screen shot of a computer code

Description automatically generated

This method was especially beneficial in situations where invalid inputs could cause incorrect or unpredictable outcomes.

**Software Testing Techniques Not Used**

While the techniques I used were effective for unit testing, there are other testing methods that I did not implement, including integration testing, system testing, and performance testing.

Integration testing focuses on verifying that different components of a system work together correctly. For example, I would test how the ContactService interacts with a database or user interface in a real-world application. This type of testing typically uses end-to-end scenarios to ensure data flows correctly between different components. Integration testing did not apply to our project milestones since we concentrated on in-memory data structures without a user interface or external dependencies.

System testing is a higher-level technique that validates the entire application against its requirements. It often includes tests for user interactions, system workflows, and overall functionality. This technique was irrelevant because the milestones did not involve a complete application with user interfaces or workflows. However, system testing will be critical in later stages of development when the application is more complete.

Performance testing evaluates application performance under specific conditions, such as heavy loads or large datasets. This could involve testing the efficiency of adding many tasks or appointments to the system. Performance testing wasn't necessary for our milestones, as we were working with small datasets and focused primarily on functional correctness rather than scalability or speed.

**Practical Uses and Implications**

Each testing technique has specific practical uses that depend on the project's requirements. Unit testing is essential for any software development project, ensuring that individual components work as intended. This technique is beneficial in the early stages of development, where the focus is on building and validating core functionality. For example, testing constraints on appointment IDs and descriptions in isolation ensures that the system behaves predictably as it scales.

Boundary testing is critical for validating user input and preventing edge-case errors. In projects involving user interfaces or APIs, it ensures that systems handle both valid and invalid inputs appropriately. For instance, ensuring that a description field does not accept more than 50 characters helps avoid unexpected errors when storing or displaying data.

Exception testing is important for creating robust and user-friendly systems. By ensuring that the system manages errors gracefully, developers can provide precise feedback to users and maintain application stability. This is particularly crucial in industries like healthcare or finance, where software must reliably handle edge cases to prevent data loss or security issues.

Techniques like integration testing and system testing become critical as development progresses. Integration testing verifies that individual modules interact correctly, which is especially important for larger systems with external dependencies. System testing is essential in the final stages of development to ensure that the entire application meets the end user's requirements. Finally, performance testing is vital for systems that handle large datasets or high traffic, ensuring they remain efficient and responsive underload.

**Assessing the Mindset While Testing**

In this project, I took a careful and detailed approach to ensure the code's robustness and reliability. It was crucial to understand the complexity and interrelationships of the code, as an error in one module could have a cascading effect on the entire system. For example, while testing the AppointmentService class, I made sure that the addAppointment method effectively handled edge cases, such as duplicate IDs:

A computer screen shot of a program code

Description automatically generated

This approach ensured invalid inputs were consistently rejected, preventing data corruption or unexpected behavior in dependent classes.

**Limiting Bias in Code Review**

To reduce bias during code reviews, I approached the code as if it were created by an external developer. This perspective allowed me to objectively identify vulnerabilities and inefficiencies. For instance, I thoroughly examined the input validation for task IDs, names, and descriptions in the Task class to ensure they met the project requirements.

A screen shot of a computer program

Description automatically generated

A computer screen shot of text

Description automatically generated

Had I approached this as my own code, I might have overlooked issues by assuming my code was perfect and moving on without proper testing. By maintaining an impartial mindset, I was able to catch and correct such potential problems.

**Commitment to Quality**

A disciplined commitment to quality is essential for ensuring the system's long-term maintainability. Cutting corners can lead to technical debt, resulting in costly fixes down the line. For example, the deleteTask method in the TaskService class was thoroughly tested to ensure it correctly handles invalid inputs.

A screen shot of a computer code

Description automatically generated

By writing comprehensive test cases, I ensured the system's reliability and reduced the risk of introducing bugs during future updates. As a practitioner, I plan to adopt tools like automated testing frameworks and maintain proper documentation to mitigate technical debt and uphold code quality.