Morgan Getkin

Software Test Automation & QA

Professor Toledo

December 10, 2023

Summary and Reflections Report

The testing approach for the Contact Service, Task Service, and Appointment Service was aligned with the software requirements. The tests cover each specified requirement, such as adding contacts with a unique ID, deleting contacts per contact ID, and updating contact fields, or adding tasks with a unique ID, deleting tasks per task ID, and updating task fields. For example, in ContactServiceTest.java, the testContactServiceUpdateFirstName, testContactServiceUpdateLastName, testTaskServiceUpdatePhone, testContactServiceUpdateAddress, and testContactServiceAddDuplicateId methods specifically address the requirement of updating with an invalid field and preventing duplicate IDs, respectively. The TaskServiceTest.java test methods, such as testTaskServiceUpdateName, testTaskServiceUpdateDescription, and testTaskServiceAddDuplicateId, address the specific requirements related to updating with an invalid field and preventing duplicate IDs.

The overall quality of JUnit tests for all of these classes is high. The tests cover various scenarios, including normal operations, updates, and error conditions. The test methods have clear names indicating the scenario they are testing, making it easy to understand the purpose of each test. The tests aim to cover a significant portion of the code. Each method in the service classes is exercised by one or more tests. The coverage percentage for all Services is now 100% across the board! The coverage percentage for the Task Service, the Contact Service, and the Appointment Service is 100% class, 100% method, and 100% line – complete coverage. There were a couple of lines that were updated after receiving feedback upon grading.

The technical soundness of the code is ensured through multiple means. In ContactService.java, the addContact method checks if the contactId is already existing. (contacts.containsKey(contactId)) before adding a new contact, preventing null references and ensuring data integrity. Updates to each specific field have their own unique methods and tests, ensuring that only required information is entered. For example, in TaskService.java, there are separate updateName and updateDescription methods to allow full control over the fields of a task object.

The provided code focuses on correctness and clarity rather than optimizing for performance, which is appropriate for the nature of the tasks (contact, task, and appointment management). Efficiency is often context-dependent, and in this case, the primary concern is readability and maintainability. Specific efficiency considerations are context-dependent and might involve profiling. In this instance, the code is straightforward, adheres to best practices, and is efficient in terms of readability and maintainability. For example, the Junit tests are named for exactly what they test for: testCreateAppointmentAppointmentDateIsPast, testCreateAppointmentAppointmentDateIsNull, testAppointmentDateUpdateDateIsPast, and testAppointmentDateUpdateDateIsNull all revolve around the appointment’s date field, and are very specific in what they do.

There are several software testing techniques that I have employed for this project. One of these is unit testing. Unit testing focuses on individual units of the code, like functions, methods, or classes. It is automated testing, ensuring quick and frequent testing during development. They test each unit in isolation, making it easier to identify and fix issues. These tests were implemented using JUnit. They were used to test every single line of code written over these three services. They ensure all of the methods and data checks are functioning as expected. Unit testing is essential for catching bugs early in development and facilitates continuous integration practices. They also accelerate development by catching issues during the coding phase and provide a safety net for refactoring.

Another software testing technique I have used is integration testing. This technique focuses on interactions between different components or systems, identifying issues in the interfaces and interactions between integrated components. It can be automated but often includes manual testing for complex scenarios. This type of testing is essential for scenarios where multiple units collaborate, ensuring they work seamlessly together, and is exemplified in testing the interactions between a class and its associative service class in these milestones. Integration testing ensures that different components work seamlessly together and helps identify and resolve issues with system interfaces. Its implementation is crucial for larger systems with multiple components, ensuring that the integrated system behaves as expected.

Additionally, I’ve incorporated negative testing into these milestones. This is designed to detect and evaluate system handling of invalid inputs or unexpected conditions, with a focus on invalid data, boundary conditions, and error scenarios. It is mostly automated for efficiency in testing various invalid inputs. Negative testing has been applied in my milestones to ensure that the system appropriately handles errors and edge cases, and ensures requirements are met. This testing method identifies vulnerabilities in the system's error handling and ensures robustness against unexpected inputs. It also mitigates the risk of software failures in real-world scenarios improving overall system reliability.

One software technique that I did not incorporate is regression testing. Regression testing verifies that new code changes don’t adversely affect existing functionality. This can be automated but may include manual testing, especially in complex systems. This was not used because there have not been any changes the code, and therefore no need for regression testing. Regression testing ensures that new code changes don’t break existing functionality, and is crucial for continuous delivery practices. It safeguards against unintentional side effects during development, enabling faster and more reliable software releases.

Another testing technique not employed in my milestones is performance testing. Performance testing evaluates system responsiveness, speed, and overall performance under different conditions. It is often automated for simulating diverse scenarios. The reason I did not use performance testing is because this program is a simple one, and there are no requirements for performance for this system, only functionality. Performance testing evaluates the system’s performance under various conditions and identifies and resolves performance bottlenecks. It also ensures that the software meets performance requirements, enhancing user satisfaction by delivering a responsive system.

Throughout this project, I adopted a cautious mindset throughout testing, recognizing the impact of overlooked issues on system stability. I also prioritized comprehensive testing to minimize the risk of undetected bugs in critical functionalities. It is crucial to understand the intricate interrelationships within the codebase. While testing the ContactService class, I considered how changes in one contact's information could affect the overall system, especially in scenarios where multiple contacts interacted. Ensuring that updates in one contact did not unintentionally impact others required a nuanced understanding of the code's complexity.

There are several ways I limited bias in my review of the code. I approached code review with an objective mindset, focusing on functionality and adherence to best practices. I strived to base my assessments on explicit coding standards and requirements rather than personal preferences. I developed and executed a diverse set of test cases to assess the code from multiple perspectives. I targeted both common and edge cases to uncover potential issues that might be overlooked with a biased perspective. For example, in reviewing the TaskService class, I consciously examined the code for clarity, adherence to naming conventions, and the efficiency of algorithms, focusing on objective criteria rather than personal coding style preferences.

There's an inherent risk of overlooking potential issues due to familiarity with one's own code, leading to confirmation bias. Developers might unintentionally assume the correctness of their implementation, impacting the thoroughness of testing. Regularly seeking external reviews and feedback from colleagues can help counteract bias. Adopting test-driven development (TDD) practices ensures that testing considerations are integrated into the development process, reducing the reliance on subjective judgment.

Cutting corners in writing or testing code can lead to undetected bugs, compromising system stability and reliability. Additionally, incomplete testing may result in software failures, impacting user experience and trust. Cutting corners to meet tight deadlines may provide short-term relief but accumulates technical debt. Technical debt can result in increased maintenance efforts, reduced agility, and a higher likelihood of future defects. Ensuring comprehensive testing in the Appointment class, including edge cases like appointments with past dates, was crucial to prevent issues related to date validation. Writing clear, self-documenting code in the Task class reduced the likelihood of future developers introducing errors during maintenance due to confusion or misunderstanding.

Regular and proactive code reviews help identify and address potential issues before they contribute to technical debt. Investing in automated testing, including unit tests, integration tests, and regression tests, reduces the likelihood of introducing defects during code changes. Maintaining clear and up-to-date documentation aids in understanding code, preventing confusion that could lead to errors. Consistently reviewing and refactoring code during development prevented the accumulation of unnecessary complexity in the TaskService, ContactService, and AppointmentService classes. Ensuring consistency in naming conventions and method structuring across all services assisted with this as well. Implementing proper error handling and logging mechanisms in the AppointmentService class contributed to the prevention of silent failures, reducing the risk of technical debt.

In conclusion, the testing approach aligns well with the software requirements, the JUnit tests are of high quality, the code is technically sound with explicit checks for null and valid field updates, and the code is efficient in terms of readability and maintainability for the provided context. The employed testing techniques align with unit and functional testing, addressing specific requirements outlined in the prompt. Adopting a cautious mindset, limiting bias, and committing to quality are foundational principles for effective software testing and development. The examples provided illustrate how these principles were applied to enhance the reliability and maintainability of the codebase.