Anabel Villalobos

August 8, 2023

CS-230

Professor Kraya

Project 2

Summary

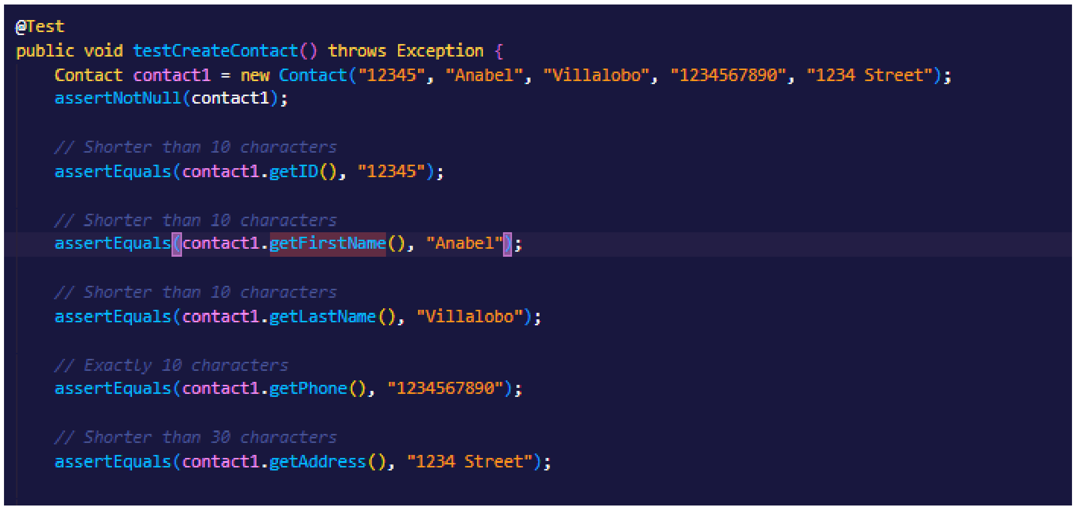
As a software engineer working at Grand Strand Systems, my primary focus revolves around delivering top-notch testing solutions for back-end software. The recent completion of a mobile application project for a client showcased my dedication to this cause, involving the development and rigorous testing of three distinct services: contact, task, and appointment. However, amidst the process, a particular challenge emerged that pertained to generating comprehensive test cases subsequent to the coding phase. The inherent difficulty in transitioning from a developer's mindset—geared towards ensuring the code's robustness—to that of a tester's perspective, where the same code's potential vulnerabilities are examined, became acutely apparent. This challenge underscored the significance of having distinct individuals involved in the development and testing phases. To overcome this hurdle, a deliberate strategy was employed: upon completing the coding for each class, I intentionally stepped away to gain a fresh perspective. This new perspective enabled me to identify potential pitfalls and edge cases that might not have been apparent during the coding phase.

The transition from contemplation to action involved several considerations. Before writing JUnit tests, I meticulously planned the testing approach. Deliberations encompassed matters such as the utilization of JUnit methods like beforeEach, afterEach, before, and after, as well as the necessity of importing specific components. Armed with these considerations, I proceeded to craft the JUnit tests for each service class. Crucially, the tests encompassed thorough evaluations of the various setters present in the task, appointment, and contact classes.

Each test was meticulously designed to encompass a wide range of scenarios. These scenarios ranged from input that was too short or excessively long to instances where the input was perfectly calibrated. My approach also incorporated extensive research, leveraging resources such as the Oracle and JUnit documentation to optimize the specificity and effectiveness of the tests. An illustrative example from the code involved the Appointment class: here, a mechanism was implemented to ensure that user-input appointment dates were consistently set in the future. This was achieved through careful utilization of the instance.after(date) method, which systematically compared input dates against the current date to prevent past dates from being accepted. This logic was thoroughly tested using different date scenarios, ensuring the robustness of the solution. I was able to use this method in order to throw an exception if the user input was anything in the past when comparing the input to today's date, which is generated at the moment the user makes their input. When I tested this in the AppointmentTest class. I used System.currentTimeMillis() to grab the amount of milliseconds for today, and added 1000000 to it to ensure a future date. When I was testing for a past input, I did the opposite. I also made sure to account for what would happen if we input the present moment.

Reflecting on the efficacy of the JUnit tests, I undertook meticulous validation. This encompassed not only confirming that the tests successfully passed but also purposefully altering the test cases to ensure that any subsequent changes to the code would trigger test failures, highlighting the tests' sensitivity to code alterations. This approach ensured that the tests were not only reliable but also responsive to code modifications.

The experience of constructing the JUnit tests proved to be enriching and educational. The comprehensive documentation and user-friendly codebase facilitated a seamless process within the Eclipse environment. My commitment to adhering to industry best practices was evident in various aspects, including the incorporation of descriptive comments within test files and consistent adherence to naming conventions. The tests were run consistently to ensure their sustained accuracy and reliability.



```
@Test
public void testCreateContact() throws Exception {
    Contact contact1 = new Contact("12345", "Anabel", "Villalobo", "1234567890", "1234 Street");
    assertNotNull(contact1);

    // Shorter than 10 characters
    assertEquals(contact1.getID(), "12345");

    // Shorter than 10 characters
    assertEquals(contact1.getFirstName(), "Anabel");

    // Shorter than 10 characters
    assertEquals(contact1.getLastName(), "Villalobo");

    // Exactly 10 characters
    assertEquals(contact1.getPhone(), "1234567890");

    // Shorter than 30 characters
    assertEquals(contact1.getAddress(), "1234 Street");
```

Beyond code quality, I ensured that the testing suite was optimized for efficiency. This was achieved by meticulously avoiding redundancy in test cases, thus maximizing the effectiveness of each test scenario.

Reflection

Throughout the course of this project, I strategically employed a diverse array of testing techniques to ensure the robustness and reliability of the developed software. This multifaceted approach included both unit testing and integration testing, each serving a distinct purpose in enhancing the quality of the software. Unit testing, a cornerstone of effective software development, involves subjecting the smallest components or units of an application to rigorous testing. In my case, this technique was invaluable as I meticulously examined each individual class and its associated methods. By isolating these components and evaluating them in isolation, I was able to detect and rectify issues at a granular level. This approach ensured that each unit operated as expected, thereby contributing to the overall stability of the software. Integration testing, on the other hand, extended beyond individual components to assess the collective functionality of the software. This technique entails evaluating how different components interact and function when combined together. For instance, in the context of this project, integration testing was instrumental in ensuring seamless interactions between services and their respective classes. An illustrative example is the successful interaction of the appointment service with the appointment class, ensuring tasks such as adding and deleting appointments were flawlessly executed. This approach not only validates the individual components but also verifies their harmonious collaboration within the broader software framework.

While unit and integration testing proved indispensable, it's worth noting that two other testing techniques—system testing and acceptance testing—were not employed in this particular project. System testing involves evaluating the entire software system as a cohesive whole. This technique is geared towards identifying any discrepancies or issues that arise from the intricate interplay between various components and services. System testing provides a holistic perspective on the software's functionality and performance, which is particularly crucial in large-scale applications. Acceptance testing, the final phase of testing, gauges the software's readiness for deployment by aligning its behavior with the requirements defined by user stories. This technique validates whether the software meets the end-users' expectations and functionalities. By simulating real-world scenarios, acceptance testing assesses whether the software effectively addresses user needs and performs as intended. Although system and acceptance testing weren't employed in this instance, they remain pivotal in comprehensive software quality assurance. By encompassing the entirety of the software system and its alignment with user expectations, these techniques provide a thorough assessment of the software's readiness for deployment.