Jordan Mitchell

10/15/2024

Professor Farley

Project Two

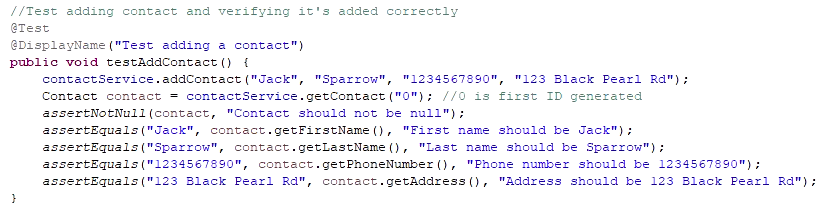
**Summary**

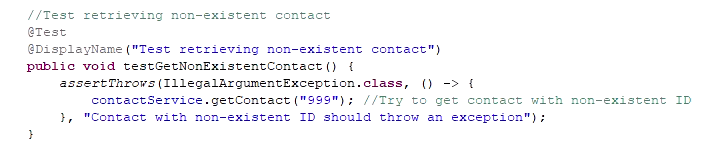
***Contact Service:***

For the Contact Service, the unit tests were to ensure that all core functionality for managing contacts worked as intended. This included adding contacts, updating contact details, and deleting contacts. This approach aligned with the software requirements by testing both valid and invalid inputs. For example:

* Valid Input: One test verified that a contact could be successfully added with a valid first name, last name, phone number, and address. This confirmed that the system accepted valid input as expected.
* Invalid Input: I included a specific test that tried to retrieve a non-existent contact that had been deleted. This verified that the service rejected the invalid input and threw the proper exception.

Below I have a snippets of code from ContactServiceTest with valid input:



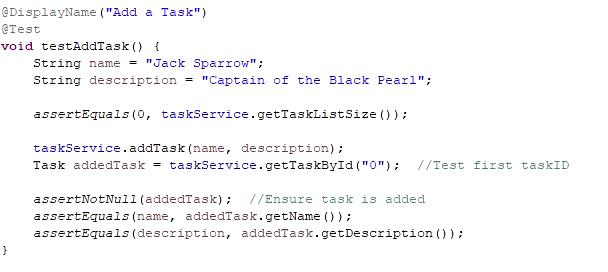
Below is a snippet of the invalid input:  


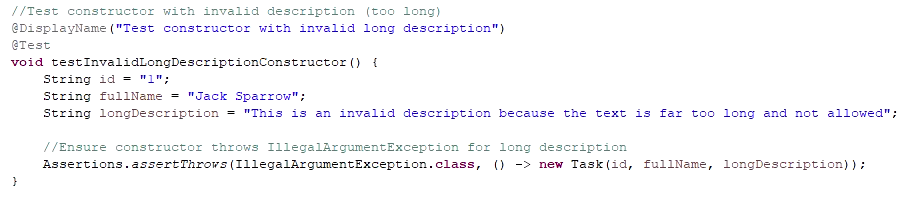
***Task Service:***

For the Task Service, my tests focused on validating task creation, updating task details, and ensuring task deletion. The requirements specified certain constraints like task names being no longer than 20 characters and descriptions being no more than 50 characters. My tests conformed to these requirements by ensuring that all fields are properly validated:

* Valid Task Creation: A test was created to ensure that tasks with valid names and descriptions were accepted and stored correctly.
* Invalid Task Update: Another test verified that updating a task with a description longer than 50 characters would fail, throwing the appropriate exception.

Valid task creation:

  
Invalid update:

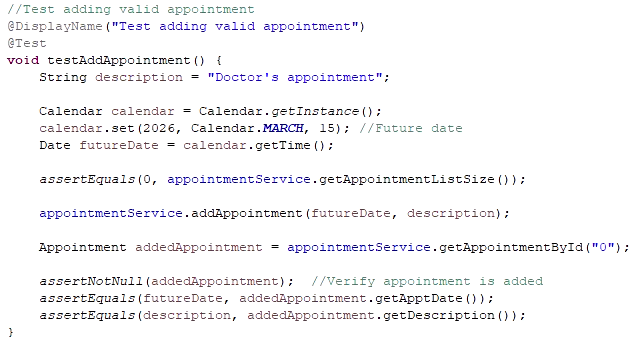


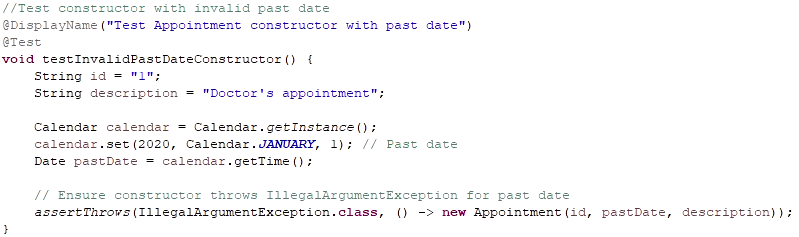
**Appointment Service:**

For Appointment Service, my unit tests ensured that appointments could be created, deleted, and retrieved based on the provided specifications. The requirement to ensure that no appointment could be set for a past date was crucial, and the unit tests reflected this:

* Valid Appointment Creation: A test validated that appointments scheduled for a future date were successfully created.
* Invalid Date: Another test ensured that an attempt to schedule an appointment for a past date was rejected.

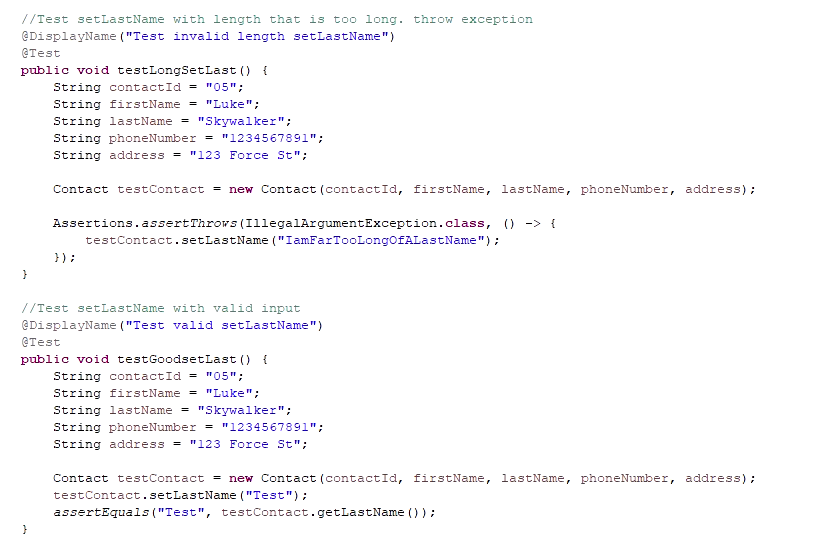
Valid appointment:

  
Invalid date:

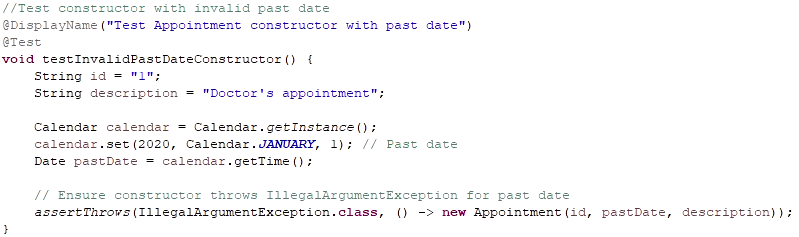
My testing approach for each service was comprehensive as I made sure to cover both valid and invalid input cases.

To ensure the technical soundness of my code, I wrote tests that validated the core functionalities and boundary conditions. Writing unit tests forced me to consider not just the "good path" but also edge cases and potential failures. Each feature was tested to ensure that it operated under both normal and unconventional conditions. For example:

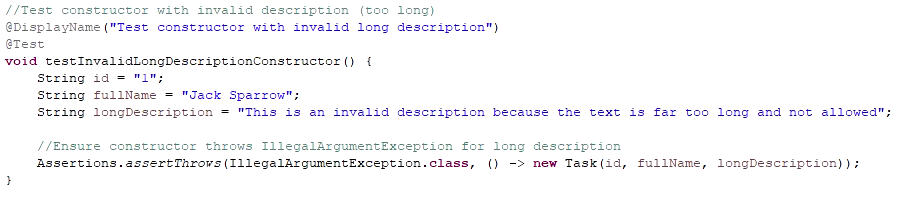
* **Boundary Testing for Fields**: For each service, I made sure to test the limits of field lengths to ensure that data integrity was maintained. For example, in the Contact Service, I tested names that were exactly at the character limit to ensure they were accepted, and also names that exceeded the limit, which were rejected.



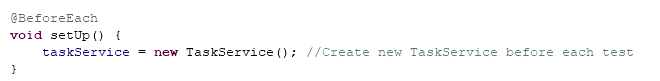
* **Validation Logic**: For Appointment Service, I included checks to ensure that appointments could not be scheduled for past dates. This validation logic was important to stop invalid data from entering the system. Below is an example from AppointmentTest that demonstrates this:

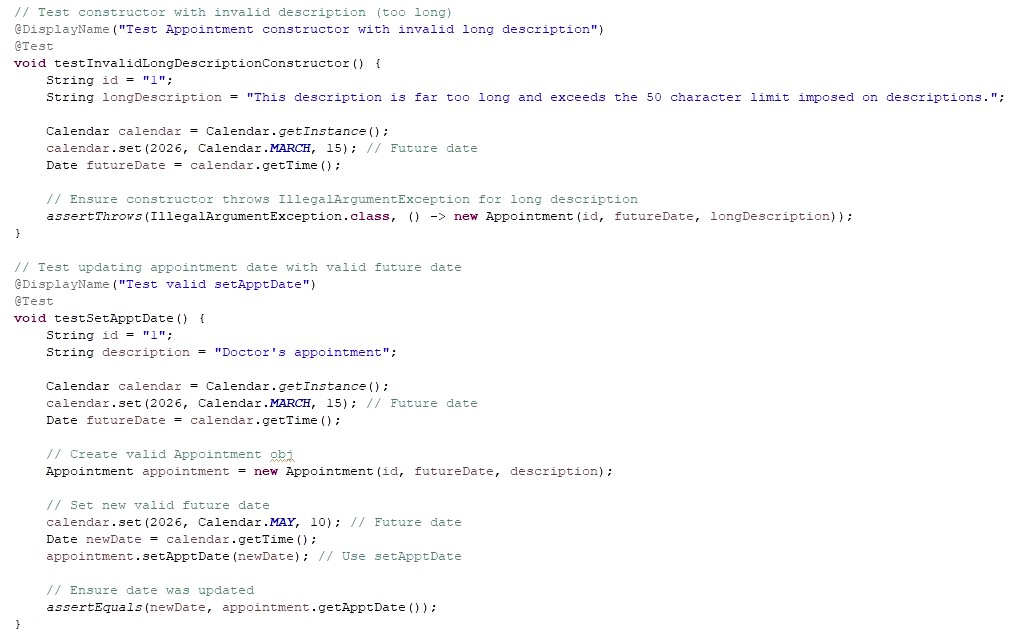
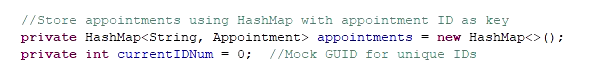


* **Exception Handling**: Another vital aspect was verifying that invalid inputs, such as overly long descriptions and null fields, were properly handled with appropriate error messages. For example, in the Task Service, trying to create a task with a null description threw a clear IllegalArgumentException that made the error traceable.

I made sure that my tests and code were efficient. I considered it from multiple angles: clarity, reusability, and performance:

* **Reusability:** In each test class, I used the @BeforeEach annotation to create setup methods that initialized objects before each test. This prevented code redundancy and allowed for a clean separation of test logic.



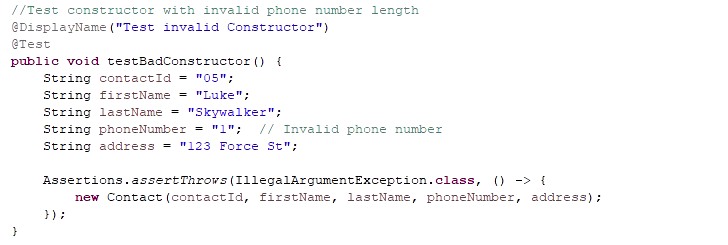
* **Efficient Testing:** I systematically ensured each test focused on a single validation issue. This kept the test cases clean and modular. For example, in AppointmentTest, I wrote clear, separate methods for testing invalid inputs such as past dates and overly long descriptions. This modular approach allows easier debugging and clear identification of specific issues if a test fails:  
  
* **Performance:** Though the scale of this project doesn’t require extensive performance optimization, I made sure the code was simple and fast by leveraging HashMaps in the **Appointment Service** to ensure O(1) for key operations like inserting, deleting, and retrieving appointments based on their unique IDs.  
  

**Reflection**

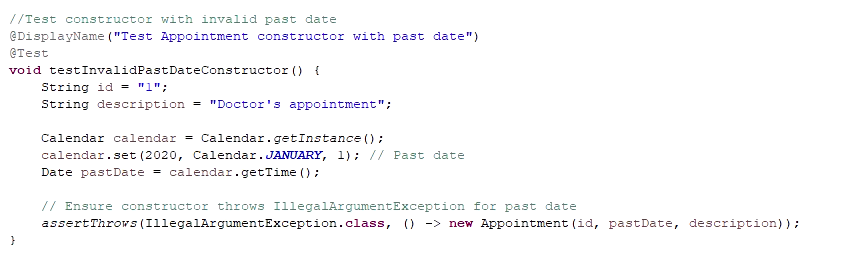
**Testing Techniques**

Throughout the entirety of this project, testing played a major role in ensuring the reliability of the code at each stage. From the early contact service to the final appointment service, my testing evolved in both depth and efficiency. The introduction of test cases helped me catch errors early, prevent regression, and gave me confidence in the functionality of each module. I often try to test repeatedly as I write my code to ensure correctness.

* Unit Testing: Unit testing was essential in confirming each method performed correctly in isolation. For example, in ContactTest, each method was tested to validate that the Contact class handled inputs like names, phone numbers, and addresses correctly. Unit tests were particularly useful for catching edge cases early, such as testing phone numbers that were too short or addresses that were too long. In the later modules, such as the appointment service, unit tests provided me with quick feedback, proving the correctness of each function.



* **Boundary and Exception Testing:** Boundary testing was an important part of guaranteeing robustness. Whether it was validating descriptions in the task service or ensuring dates were in the future in the appointment service, these edge cases were vital. For example, in AppointmentTest, testing invalid past dates or long descriptions helped prove that all potential issues were handled correctly by the system:



As shown, from the contact service to the appointment service, my testing techniques evolved and became increasingly comprehensive, making sure each component behaved as expected. Testing was integrated into every aspect of the project, ensuring that the functionalities, ranging from simple field validation to more complex exception handling, were thoroughly checked before moving forward. As the project grew, testing evolved from simple unit tests to include more sophisticated testing approaches making the overall process both more efficient and effective. The challenges I faced throughout, particularly with handling edge cases, were solved by writing thorough test cases. Debugging had become more focused as testing grew more comprehensive. A specific challenge was managing edge cases, such as testing for past dates, which was solved by ensuring that tests captured a variety of scenarios, including valid and invalid inputs.

**Mindset**

When I first started this project, my initial mindset toward testing was primarily focused on making sure my code worked as expected. I thought that writing a few tests to verify the main functions would be enough. However, as the things progressed, I realized that testing wasn’t just about proving the code works but also about catching and preventing potential problems early, handling edge cases, and making sure that the code would still work as it evolved. I realized that without proper testing, I was often revisiting the same bugs, wasting time fixing issues that could have been caught earlier. Testing soon became less of an afterthought and more of an integral part of my coding process. For instance, in the contact service, I initially tested basic functionality manually. This worked for a while, but as the project grew, manual testing quickly became inefficient. When I implemented JUnit tests for ContactTest, it opened my eyes to how testing could shorten my development process. It wasn’t just about ensuring things worked, it was about screening potential failures before they even surfaced as the system grew more complex. Testing made my code more reliable and ultimately saved me time in the long run.

I realized the importance of testing each aspect carefully. One example is the appointment scheduling system, where I had to validate dates and ensure that appointments could only be created for future dates. This required extra caution since the logic for date comparison had to handle both edge cases and user input errors. Testing invalid inputs, such as past dates or overly long descriptions, was crucial to ensuring the integrity of the code. I've grown a deeper understanding of the complexity and interrelationships at work. For instance, AppointmentService depended on the correct behavior of the Appointment class, meaning any failure in the validation logic of Appointment could lead to larger issues in the overall functionality. This required me to be cautious and intentional when making any changes, as small errors could snowball into larger issues. By recognizing the interconnectivity, I was better able to anticipate where failures might occur and create tests to catch these early.

One challenge I faced was limiting my bias when reviewing my own code. It’s easy to fall into the trap of assuming that your code works because you wrote it, or that it's "good enough". To remedy this, I deliberately wrote test cases that aimed to break the code, rather than confirm it worked. For example, in the Task class, I wrote tests that pushed the limits of validation, such as checking how the system handled extremely long names or invalid inputs for descriptions. By doing this, I stepped outside my developer mindset and focused on potential failure points that a user may encounter, which helped me avoid mistakes I might have otherwise overlooked. Bias can be a problem when testing your own code because it's easy to have blind spots. I knew how the code was “supposed” to behave, but that also meant I could overlook potential edge cases that might break under different circumstances. At first, when validating dates, I only tested obvious cases before I later realized that there were edge cases like appointments on the exact date boundary (e.g., today vs. tomorrow). My original code didn't account for this, and I was able to break my code repeatedly, before ending up with my final product. This approach helped reduce my bias and led to me creating more comprehensive coverage.

By the end, I learned to appreciate the importance of being disciplined in my commitment to code quality. It was tempting at times to cut corners, especially when the code seemed to work. However, I understood that doing so would likely lead to bad habits and technical debt that might not show immediately but would cause problems down the line. In the project, testing the deletion of appointments was an important requirement. Failing to properly delete or attempting to delete non-existent appointments had introduced inconsistencies, so I committed to ensuring these scenarios were fully covered by robust code that could handle multiple kinds of input. To me, avoiding technical debt means being thorough during both the writing and testing phases. Skipping tests or leaving validation unchecked can lead to more work later when bugs arise in production. Thinking like a user is another great mindset to adopt when facing this. I'm reminded of video games where developers release a polished game, only to find a player has realized how to break past the bounds of the game by doing something completely illogical, yet possible. I plan to maintain a habit of writing robust, modular code that’s followed by thorough unit tests. I’ll also continue using practices like code reviews and testing edge cases to ensure I don’t introduce unforeseen issues into the codebase. This will help me stay disciplined and deliver reliable, maintainable code as I progress and advance in my career.

In conclusion, this project has significantly shaped my understanding of software testing and development, emphasizing the importance of repeated testing, thorough validation, and a disciplined approach. As I progressed through each module, I experienced firsthand how proper testing practices lead to time savings, reduced errors, and more reliable software. By embracing techniques like edge case testing and anticipating potential failure points, I was able to create a more resilient and maintainable product. I’ve also come to appreciate the interconnections between different components in a system, recognizing that a seemingly small bug in one part can have a ripple effect elsewhere. Moving forward, I will continue applying these lessons, staying committed to writing efficient, well-tested code while keeping an eye on preventing technical debt. I've grown fond of code modularity, appreciating how much more efficient debugging becomes as a result. I've both refined my coding abilities and built a stronger foundation of the discipline it takes to build quality software that can stand the test of time.

Resources

*Technical debt*. (2021, September 2). ProductPlan. <https://www.productplan.com/glossary/technical-debt/>

*What is Unit Testing? - Unit Testing Explained - AWS*. (n.d.). Amazon Web Services, Inc. <https://aws.amazon.com/what-is/unit-testing/>

Radigan, B. D. (n.d.). *Tech Debt Signs & How to Effectively Manage it*. Atlassian. <https://www.atlassian.com/agile/software-development/technical-debt>

SonarSource. (2024, April 3). *technical debt developer’s guide*. Sonar. <https://www.sonarsource.com/learn/technical-debt/>