Danica Hesemann

CS 320

February 21, 2021

Summary

My unit testing approach was very similar for each of the three features. I was very intentional about testing each individual requirement. I read the requirements, deconstructed them into multiple smaller requirements, and accounted for each one, creating test cases wherever possible. For example, one requirement was: “The contact object shall have a required unique contact ID String that cannot be longer than 10 characters. The contact ID shall not be null and shall not be updatable”. I broke that down into five smaller requirements. First, the contact ID string is required. To meet this requirement, I incorporated the contact ID into my Contact constructor and used that constructor for my function to add a contact. Second, the ID must be unique. I created a test case for this by adding a contact to the list of contacts and calling the addContact() function using the same contact ID. Third and fourth, the contact ID could not be longer than 10 characters and could not be null. I created unit tests for both of these requirements by creating two test cases, testIdTooLong(), which tried adding a contact with an ID that had 11 characters, and testIdNull(), which tried adding a contact with an ID set to null. I made sure to have a test case with a 10-character ID as well, to make sure the code would only throw an exception with 11 characters or more. I am confident in the effectiveness of my JUnit tests based on coverage percentage because even though the overall coverage is only 81.9%, each of the service classes (ContactService, TaskService, AppointmentService) each have 100% coverage. The Contact, Task, and Appointment classes do not have 100% coverage, but I have examined them and found that the only parts that were not tested are a few mutator methods.

Since there was no interface and I could not ensure the code was technically sound by running it, I used JUnit tests to ensure my code was working properly. I reviewed my code and corrected any errors or warnings that I could see, then I used JUnit testing to make sure every requirement was met and the code was behaving the way I expected it to. This proved to be a useful strategy, as the first time I ran my tests for the ContactService class, there were several errors and failures that I had to fix. I also analyzed the coverage percentage to ensure I had not missed anything. For example, when I expected to see 100% coverage for the ContactService class it only had around 94% coverage so I reviewed the code and found that the function for updating the last name was not tested. I figured out that I had forgotten to change a unit test that I copied and pasted from the function to update the first name, so the test was executing without errors or failures, but that test was testing the wrong thing.

To ensure my code was efficient, I built things like length requirements into constructors or functions using conditional statements wherever possible. For example, before creating an instance of Task, the code would throw an exception if the task ID is greater than 10 characters or null. I also created a function for each service class to search for an ID, since I needed to do that for several of the functions. I annotated my code clearly and concisely so that it is understandable and easy to read. I also chose to set up and tear down tests using @BeforeEach and @AfterEach and built my unit tests around the same setup instead of setting up tests individually.

Reflection

For this project, I reviewed my code and used unit testing by creating JUnit tests. JUnit testing allows a developer to make assertions about how the code should behave and test whether it meets expectations. This allows developers to check that the code is working correctly and also that it responds appropriately to any incorrect input. For example, I may check that addAppointment() works correctly with expected input, then test it with an ID that is too long and assert that it will throw an exception.

Additional testing techniques that I did not use include static testing techniques such as peer reviews or inspections, and dynamic testing to test the security, performance, or behavior of the overall program. Static testing can even be done before any code is written by reviewing documents and ensuring that the design meets the specified requirements. Once code is written, it offers helpful feedback about what may need to be improved. Peers or moderators may be able to catch bugs and errors in code that you missed, failures to adequately meet requirements, or other issues that may make the code or design more efficient. It is also important to perform tests that involve executing the program because it may be necessary to ensure the interface looks good, that tasks take a reasonable time to execute, and that every function works as intended in its context to meet the system requirements.

Acting as a software tester, I tried to adopt the mindset of someone who wanted to find all the potential flaws in my code. I tested for everything I could think of and analyzed the coverage to find every problem. One example from my experience that highlights the importance of the complexity and interrelationships of the code would be that at one point I was writing a test to see whether the deleteContact() function worked properly and I had changed part of my setup, where I had been testing a list with two Contact object in it and I had changed it to testing a list with one object. However, I had neglected to account for this change when I was checking whether the function worked. I was checking if there was a length of 1, which was not working because the list was empty after deleting the single object. It also would have claimed to work correctly if the function failed to delete the object.

It was hard to limit bias since I was painfully aware that I wrote the code with certain requirements and challenges in mind and it was difficult to detach and try to imagine additional problems to test for. I tried to carefully think through the potential issues and to learn from problems I came across that I had not anticipated, but I still am unaware of whether there was more tests I could have conducted. For that reason, I think bias would be a concern if I was testing my own code. From past experience in coding classes, it helps to have another perspective (or several perspectives) to point out anything I may have overlooked.

It is extremely important to commit to quality when writing and testing code. It can be tempting to stop when it seems sufficient, but without thorough and careful examination and testing, the code could take much longer to finish or it could be released with bugs that could have serious consequences. I intend to avoid technical debt in the future by carefully reviewing my code and having a peer look at it and make suggestions as well whenever possible. I will also try to develop efficient code that can be modified relatively easily in case I find a problem and need to rewrite. I understand the importance of writing quality code and the potential cost of cutting corners, so I will test my code using whatever resources I am able early and often to avoid serious problems in the program later.