CS 320

Roger Newton

Summary and Reflections Report

Throughout this course, I have learned a lot about unit testing and how it fits into software development. Before starting this project I had very limited experience writing test cases or using JUnit. When I began working on the Contact Service I struggled to structure my test cases properly and even had trouble running them at first. I originally tried using VS Code but ran into file path issues that made testing nearly impossible. After switching to the Eclipse IDE the process became much easier because of the built in JUnit testing and then I could focus on creating better code rather than fighting to compile correctly. It also helped me shake the rust off working with Java because it's been a while since I used it. Once I had my environment set up correctly I created dedicated test files for each service class to make sure that all the methods and requirements were being properly validated. The pattern I followed was to write specific and small test cases that verified one functionality at a time. This helped show me that if a test failed then I would immediately know what part of the program was not working correctly. Each week I built off my previous code and testing techniques to create more efficient and reliable tests.

My testing approach for each of the three services(Contact, Task, and Appointment) was based on matching each test to a specific software requirement. The Contact Service required unique IDs under ten characters, first and last names under ten characters, a phone number exactly ten digits long, and an address under thirty characters. All of these fields also needed to be not null. To meet these requirements, I created specifix tests like (testContactIdTooLong())

and (testContactIdNull()) which clearly labeled what they were verifying. Naming my tests in this way made it much easier to know what each test covered and helped check off all the assignment requirements. When I moved on to the Task Service the requirements were pretty similar in structure, but it had different character limits for names and descriptions. Since I had already developed a strong foundation for how I wanted to organize and write my tests I was able to reuse some past testing patterns. This reusability saved me time while also keeping my tests consistent. After reading through my submission feedback, I took the opportunity to improve where my earlier assignments were lacking. This was mainly done by adding more edge case testing and validating error conditions that I initially overlooked in the Contact Service.

The Appointment Service introduced some new challenges with the date validation. I had to make sure that appointments couldn't be scheduled in the past. To do this I used Java's Calendar class to generate test cases for both past and future dates. For example, I used (calendar.add(Calendar.YEAR, -1)) for a past date that should fail validation and (calendar.add(Calendar.MONTH, 1)) that shows valid future date that should pass. I also tested for duplicate IDs and verified that the delete functionality worked correctly. In the tests I would delete an appointment and then call the delete method again to make sure it was actually removed. I figured this would be the best way to make sure a deleted ID wouldn't somehow stick around.

Overall I think that my testing strategy was aligned with the project's requirements. I made sure that all the rules in the requirements document had at least one matching test case. My JUnit tests were effective because they included both valid and invalid input cases. This allowed me to check that the program handled all the possible outcomes. Using the coverage tool built into Eclipse helped me confirm that I was getting good coverage across my classes. The tool

highlighted tested code in green and untested code in red. Once I figured this out it was a game changer. I am more of a visual learner and the highlighting made it alot easier to guide me to where I need to focus my attention. One of my favorite tests was (testAddDupContact()) which checked that trying to add two different contacts with the same ID would be invalid. This was a simple yet satisfying test because it confirmed that I was meeting the requirements and it was working properly. I also tested shortened and lengthened phone numbers with the same logic. I tried to stay consistent with my testing methods in all the files. This approach gave me confidence that my tests covered all requirements.

The main testing technique I used was unit testing to focus on individual pieces of code. Keeping the tests small and specific made it easier to find problems quickly.The other notable testing technique I used was boundary value testing to make sure my validation logic worked. I did this in the Contact Service by testing phone numbers that were both too short and too long to make sure errors were thrown. Then in the Task Service I tested name and descriptions that were right at or just over the character limit. Lastly in the Appointment Service, I tested the dates to confirm that past dates weren't accepted and the future ones were.

There were several testing stylesI did not use in this project, mostly because they weren’t necessary for the assignments. One of then is integration testing for checking how different services interact with each other. Since all the Services were designed to work separately the integration testing didn't apply. Another technique I didn’t use was system testing which focuses on testing the full functionality of a system from backend to frontend with the user interface. We didn’t have a UI for these projects so that testing just doesn't apply. Lastly I didn’t use performance testing because the project didn't require it and we were working with a small amount of data. Each of these techniques have their own place in different scenarios. Unit testing

is especially useful during early development for catching errors before they get harder to find. Boundary testing is great when an app has strict data limits. Integration and system testing would be more important in a larger project that has all services connected together. Performance testing would also become important to make sure the app runs efficiently when there's a lot of users or large amounts of data.

Throughout this project I tried to maintain a cautious mindset and analyse the best way to approach problems. I realized early on that even small mistakes within my validation logic could lead to bigger problems later. By testing not just the normal scenarios, but also the invalid ones I learned how critical it is to consider every possible form of bad data. For example in the Appointment Service, I tested future dates, null dates, past dates, and dates that were close to the limit. This helped me catch subtle issues that might not show up with normal usage but could possibly still cause bugs. The other shift i noticed was when I realised how important it is to understand how different parts of the code interact.

Another big lesson for me was learning to limit bias. Since I am acting as both the developer and tester, it’s natural to assume that my code works and only confirm that without looking from a different perspective. To avoid this I wrote tests that tried to break the program on purpose. An example of this is in my (testPhoneBadLength() test. I used a 7-digit number like (8675309) to confirm that the validation logic would catch the error. Writing tests this way helped me think critically about what could go wrong instead of assuming things would only go rightt. If I were to review someone elses code I would probably be even more critical than I was of mine. I could see this being a problem when having the developers test their own code.

These three service assignments taught me more than just how to write tests. They helped me understand the value of testing during the entire development process. When I first started I was rusty with Java and unsure how to even use a JUnit test. By the end I was writing organized test cases that directly covered the software requirements. Each milestone helped me refine my code, test cases, and build confidence in my development skills. This project gave me first hand experience that I can take into my future job or projects. I now understand how important it is to write maintainable tests to validate normal and edge cases. Overall, this experience helped me grow my development skills and give me a taste of writing test cases like most developers are required to do now.