Ashley Santo

Project 2

Summary

I think my testing approach aligned very well with the software requirements. I made sure that for each requirement, there was a test to verify that requirement was being met. For instance, with the character limit requirements, I would make sure that each item on the list was being tested for the length and making sure it would throw an exception when it would go outside the set boundaries. I was also testing to check that the contacts/tasks/appointments were being input into the system correctly and that all the items were being added, deleted and updated accurately. I did this by having the JUnit tests confirm the true or false conditions I set as well as making sure exceptions were being thrown correctly. I know my JUnit tests were effective based on the coverage percentages. The Task Service, Contact Service and Appointment Service classes had a coverage of 95% while the Task, Contact, and Appointment classes had almost 82% coverage.

I was able to make sure my code was technically sound and efficient through the tests that I ran. By using both true and false statements, I was able to confirm through JUnit tests that the program was functioning correctly. An example to illustrate is the test starting at line 32 of the TaskServiceTest.java file. I added three tasks with unique IDs and then ran two checks to make sure it was updating the tasks correctly. In line 43, I’m updating the task connected to the ID “0000000001” to “playing” which should return true since that ID is one of the three in the list. In the following line, I’m trying to update the ID “0000000012” which is not an ID in the list to “playing”. This should return false since the ID doesn’t exist. Both passed the tests. Also, using proper naming and syntax for classes and methods helped ensure all the code was being run properly. For example, with using “@Test” so the tests would be read properly or using assertions to make sure exceptions are being thrown such as in line 21 of the Task Test class.

Reflection

For this project, I mainly used structure-based techniques. Structure-based, or white-box, techniques are where the internal structure or design of the program is known to the tester. Usually, developers are the ones implementing these tests since coding knowledge is needed to understand how the program is performing. This is seen through the JUnit tests where I’m verifying that Boolean true and false statements are accurate, that exceptions are being thrown when needed and that the program is adding, deleting and updating information correctly. All of the testing I’m doing is internal to the program based on the code written.

The other software testing techniques I didn’t use for these milestones is specification-based and experience-based techniques. Specification-based, or black-box, techniques check for both functional and non-functional aspects. This is where test cases are derived from some specification or design of what the system should do. It doesn’t check for how the system gets there, as long as the desired behavior is appearing. Usually, testers have no knowledge of the code behind the scenes and give the perspective of how end-users react to the product. Because I’m aware of the internal structure of the programs, this wasn’t implemented. It’s also because I was using tests and coverage to make sure the code is functioning correctly. I was never looking at just specifically the behavior of the program as an end-user might. The other technique I didn’t use is experience-based techniques. This is where testers use their previous experience to determine which areas of the system are the most important to focus on and areas that are expected to be used the most or are most likely to have more errors. This comes from having experience with similar products or systems and then using that to help with future systems. Because I’m not very experience in testing or coding in general, I didn’t implement this since I’m still learning what to look for and where errors may show up.

Each of the techniques I listed above have their own practical uses that may change depending on the project and what’s necessary. Black-box techniques are for tests to give a more end-user perspective and focus more on what behaviors are showing rather than how the program is doing it. There doesn’t need to be any coding knowledge or awareness of internal structures. With white-box testing, the tester or developer needs to know the internal structures of the program and need coding experience. This is to make sure the internals or designs are accurate. Lastly, experience-based techniques are rarer cases to use but come in handy in situations where there may be time constraints stopping structure-based tests or there aren’t enough adequate specifications to get accurate results for white-box testing. Testers or developers can use previous experience to focus on areas they are aware may have issues or may have more traffic from end-users.

The mindset I tried to adopt when working on this project is patience and focusing on the goals of each portion. It can get easy to get lost and frustrated with coding so I like to break it down piece by piece and focus on what this specific part is supposed to do. I tried to employ caution when it came to testing and trying to think of anyways the code could be exploited or defective. As the developer, it can be very easy to think that the code will function correctly and overlook any issues that might occur. So, I tried to think of anything that could cause an issue to arise and then test for it to make sure the program responds correctly. An example in my project is that I used many true and false statements to make sure that if something is outside the guidelines or isn’t a valid input, that the program will respond accordingly. I think limiting bias goes hand in hand with using caution. I didn’t want to just assume that the code would work correctly and tried to test every problem that could come up from it. I can definitely see how bias would be a major concern when testing your own code. I’m still relatively new to software development so I’m more cautious in realizing my work may have more errors or bugs. For example, I was having issues with the date being read incorrectly for the appointment sections and needed to go back and figure out what was going wrong. But I can imagine that as someone gains experience, they might become more relaxed around testing their own work. This kind of overconfidence can lead to defects being released unknowingly and causing anywhere from minor to major problems down the line. That’s why it’s important to keep biases out of it and test your own work as you would someone else.

It’s important to be disciplined in the commitment to delivering quality code because cutting even the slightest corner can have drastic results. It’s important to make sure we are writing the code securely and that it is being tested for every issue that could possibly occur. In order to avoid technical debt in the field, I think I would try and treat my code as if it’s my first time looking at it and keep biases out of it. That way, I can sort of look at as the end-user and figure out what they might do to make sure everything is working properly. As well as testing what would happen if the end-user was to reach outside the limits and expectations of the program.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). 4. Test Design Techniques. In *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide* (3rd ed., pp. 77–129). essay, BCS The Chartered Institute for IT.

Nowacka, E. (2021, November 3). *Black-box and white-box testing - pros and cons*. Grandmetric. Retrieved February 20, 2022, from <https://www.grandmetric.com/2021/03/11/blackbox-whitebox-testing/#:~:text=Black%20box%20testing%20is%20considered,of%20the%20software%20in%20question>.