Name: Cody Stuart

Course: CS-320 QA Automation

Assignment: Project Two

This report summarizes the unit testing approach, experiences, and reflections related to the development and testing of the Contact, task, and Appointment services for the Grand Strand Systems mobile application.

My approach for the unit testing for this assignment was simple. For each class I would break down the customers’ requirements and write a JUnit test for each requirement. For example, the contact, task, and appointment service all require a unique ID. This meant my testing needed to validate that not only could an ID be set, but that another entry into the service could not have the same ID of another previously entered item. I did this for each requirement provided I wrote a list of what each assignment called for unique IDs, character limits, etc. Character limits were another customer requirement to pay attention to, some of the fields I was told could be no more than 10,20, and 50 characters. This meant validation needed to be done at the method level to throw an error when a value exceeded what was allowed. With the testing of these methods, the attempt would be made to add to add an item where the specific field being tested had too many characters. The description field, for example I used I believe 63 characters where the max is 50 and successfully threw an error. I’m extremely confident in the overall quality of my tests as well. I ran the tests multiple times and read them over to confirm they were testing what it was that I wanted them to test. Additionally, JUnit provides the option to calculate the code coverage, and with these assignments I was able to achieve 100% of code coverage for both my application and service classes. This is true for all three contact, task, and appointment coding portions. Given that the required coverage was only 80% I believe that the 100% coverage is a measurable vote of confidence in the thoroughness of the testing, especially since the test were all successful as well. So, with the test being written based on the requirements, the code being 100% covered, and all of the tests passed I’m confident that my tests prove my code meets the customers’ requirements.

To reflect, unit testing was an obvious testing technique used in this scenario, of which I just discussed. Each individual class was tested based on the requirements set. The unit testing focused on independently testing each class’s written methods. I took advantage of JUnit functionality such as the different kinds of assert, which affirms the response is the expected. As well as @BeforeEach, which does some setup in this case, each unit test for the service classes started with a clean hash map. There were also elements of black-box testing, where the input and output of methods being tested against the expected results was done, rather than the internal code implementation. Like the unit tests, the black box testing is also derived from the requirements and makes sense that these two methodologies overlapped here. My testing also asserted that exceptions were thrown when expected. Such as when a unique ID was more than 10 characters. Expecting an exception, let us know when the program is not correctly throwing an error. These errors should be concise, yet descriptive of the issue, and avoid any information that could be used to attack the application. Exceptions were expected to be thrown when certain fields were found to be null, like an ID or a name. As well as when appointment dates were selected from the past. Thank fully JUnit provides assertThrows to make this testing extremely easy to validate. A testing methodology not used for these purposes was integration testing. Integration testing would have confirmed the successful interaction between different software components, such as when validating that an online store inventory works with ordering and payment systems. Had I connected the task, contact, and appointment services together, integration testing would have been warranted. Given that this code is for a mobile application performance testing also would have been valid to be performed on the written code. Again, though this type of testing was not used. Considering the code was not connected into or as a part of a larger software application, performance testing would be early. There may be benefits to performance testing components individually as well though.

Each of these types of testing has their own unique benefits, unit testing for example allows us to catch bugs early in development and provides a concise location as to where the issue lies. For example, when I ran some of my early testing before submitting my assignment for the appointment service, I discovered I had flipped my date check. So, when I asserted the date was after today, I always failed because the method itself was validating that the date was before today. Black box testing, unlike unit testing, doesn’t require internal knowledge of the code, and focuses more on requirement validation. This is best used in acceptance testing when the customer is ready to receive the product. Exception testing, as discussed earlier, leads to more stable and reliable applications. It serves to improve input validation and error information. Integration testing, which was not used, is essential for modular systems and systems built from multiple components such as a Kubernetes deployment that writes to a stateful database.

For the testing, it is important to consider a multitude of factors when writing your tests. Caution is one of these factors, for example throwing an exception when a field is null allows us to think like an adversary who might attempt to pass null values into the program in order to act maliciously. Additionally, properly throwing an exception can assist in preventing un-necessary crashes to the code. This pre-emptive error handling provides the caution we need to maintain the program in a running state. Due to the dependency of the contact service on the contact class it was also important to write JUnit tests for both. Otherwise the testing itself would not have been thorough enough. I believe I’ve covered bias enough without specifically stating that I was limiting bias. With my testing I focused heavily on the provided requirements and less on how the code was written. With the black-box style of testing the requirements are the key, the underlying written code is either unknown, or unimportant to the actual test being performed. Bias is still a concern and should be considered and checked for when possible. There are assumptions that could be made by overlooking tests for edge cases. I could have also written my unit test just to make sure my code works rather than writing code to test the requirements. Discipline is another important, and arguably more important than bias, and caution. If you have discipline you can push yourself to be cautious and unbiased. I will say that some of the better software engineers I have met have embodied discipline in other parts of their lives as well, such as constantly pursuing personal learning, and using their spare time to work on personal projects. As well as following strict plans around physical fitness. Following this discipline also means not cutting corners when it comes to writing or testing code. Cutting corners could cause input to not be validated or security to not be implemented. Imagine if the incident involving the Ariane 5 rocket where integer overflow was unaccounted for had people on it. This improperly vetted code could have cost lives. In a world like today SpaceX rockets are extremely technical and I could only image the amount of security that must go into this code. What if hackers could find a way to remotely manipulate the rocket and cause grave things to happen? There are serious implications to not being thorough with your code. An effective way to avoid technical debt in this field is to target high, and meaningful test coverage. As well as writing tests concurrently to the code.