**7-2 Journal: Project Summary**

Wayne A. Silva II

Southern New Hampshire University

CS-320 Software Testing, Automation QA

Instructor Desmond Springer

December 22, 2024

My testing approach was requirement driven and specifically focused on unit tests, boundary tests, and exception handling to ensure that data was validated and handled according to the provided requirements. Testing coverage was also a significant concern as a coverage threshold was provided. I leveraged established helper methods such as assertThrows() and assertEquals() in my testing for consistency. My approach was well aligned with the requirements of the project and its features. I wrote tests based upon requirements such as input length, scope, functionality of features, and error handling. In the appointment service I wrote tests that checked if appointments were added correctly and if duplicate appointments had been added.The test functions testAddAppointment(), testAddDuplicateAppointment() are examples of this. In the contact service, testAddContact() creates a new contact passing in generic user data to validate whether the contact is successfully created. The assertEquals() helper method was used to check if the provided contact matched the created contact. The overall coverage of my JUnit tests was strong. There was definitely still some room for improvement, however. Critical logical paths were tests based on requirements and edge cases were considered. The tests were written based on the requirements directly and covered many important areas. This supports my claim that my tests were effective. My overall test coverage was a little under the threshold, but, overall they were pretty comprehensive. A second pass would have allowed me to bridge the gap between my actual test coverage and the coverage threshold. Requirements such as ensuring valid input length and type, checking core functionality, ensuring no duplicate entries, proper error and exception handling, and boundary testing null input were tested. The tests written were meant to cover all of this criteria which was outlined in the requirements document.

To ensure that my code was technically sound I adhered to best practices, debugged my code, relied on helper methods when possible, and carefully considered the requirements of the project. I began by reading through the requirements for each milestone and considering what functionality was needed. For each functional requirement, I created test cases that checked that the method or input behaved as intended. Additionally, I considered test cases where unexpected functionality or input may occur. A significant challenge was really thinking through all the edge cases. In development, it can be difficult to conceptualize all the ways a user might use a feature. I tested for simple edge cases based on the requirements of the project. Having access to reusable helper methods helped with test consistency. I focused on input validation, exception handling, and core logic to ensure that the code was technically sound. Ensuring that code was efficient was made possible by the inclusion of reusable helper functions and also by creating instances of classes such as new tasks and new contacts which could be used later in the test. In this project, I conducted unit tests, boundary tests, and exception handling. Unit tests made up a significant number of the overall tests written. Many of the unit tests were written to check that the functionality of public methods were aligned with the requirements of the project. For instance, testUpdateContact() created a new contact service and contact with generic parameters. Then it called the addContact() method passing in the newly created contact. From here the test updated contact information and then called assertEquals() to check if the newly added information was the same as the returned information. Many of the tests written followed a similar format. Boundary tests focused on testing the validity of input. As an example, testInvalidName() used assertThrows() to test if a null name field and a name field with a string that was longer than the required character length would throw an illegal argument exception. An example of an exception handling test istestDeleteNonexistentAppointment(). This test creates a new appointment service and then attempts to delete an appointment from internal memory with an ID that does not exist. The test again uses assertThrows() to check that an illegal argument exception is thrown. I did not conduct integration or performance tests in this project. Testing that each feature integrates with a broader system or with other features was beyond the scope of the project. In a production setting, working with large codebases and developing new features would likely demand that integration tests are carried out to test if the new code will integrate with the existing code. This could also be automated if needed. The project used internal memory stored in a hashmap for each service. While performance testing was not carried out this time around it could be beneficial to conduct tests that analyze how the system performs under load. Perhaps the project will be scaled up over time.

While working on this project I initially started out in the mindset of a developer trying to produce code that aligned with the provided requirements. About the time that I began writing tests I realized that a mindset shift was in order. It was not just about what work products I produced but how well I understood the requirements and my ability to think outside the box to test that these requirements were met. Caution was required when considering edge cases. There are many ways that something can go wrong or a user can interact with a service in a way that was not initially intended. Sometimes a browser window is reloaded or a user has to refresh their window. In cases like this it is possible that user data is submitted twice or perhaps even re-submitted by the user. The test method testAddDuplicateAppointment() checks to see if when a duplicate appointment is entered that the system picks up on this and throws an exception. I honestly did not spend too much time considering bias when reviewing the code. I can see now how writing code could potentially lock me into a particular way of thinking. Because of this, I may not be able to look at the project or feature comprehensively and objectively. It is important to be disciplined when writing tests. This is because the testing process ensures that the program will behave in a way that is aligned with testing objectives and user requirements. Writing code with as much attention to detail as possible and thinking outside the box can really help to mitigate risk and liability later.