Summary and Reflections Report

Quinlin MacKenzie

Southern New Hampshire University

CS 320: Software Test Automation

Johnathan Norman

23 June 2024

**Summary**

In the development of the contact, task, and appointment services, the software requirements were mainly limited to verifying length of the input fields and confirming the information being input. Each set of services have the main object class for creating objects, which checks for exceptions on creation and have getter and setter functions for updating the object, and the service class for managing objects of a certain type, allowing us to set up checks to throw exceptions whenever objects are added, removed, or altered within the service. The test cases were then set up to test for each exception case that we created when writing the main classes. This is where many of our requirements were verified as being met, such as testing the upper and lower limits of our input fields for invalid information or comparing the date information provided in the appointment service to the systems date information. Overall, the JUnit tests written cover approximately 82% of the written functions. This is sufficient to meet industry standards, but we could also do more work to cover some of the gaps in our formatting, such as confirming the format of date information instead of simply comparing against the current date. We do cover the necessary functions in the JUnit tests though, ensuring that the initial states of each input does not throw the coded exceptions when they are supposed to.

In terms of managing the efficiency and technical structure of the code, some feedback was needed and adopted into the project in specifically creating new objects. Originally, my task service did not update tasks, but instead created new objects each time a detail needed to be changed. Going back and adding a series of getters and setters to the task class was the first step in ensuring efficiency in the code, and allowed me to simplify many of the test cases, which led to less resource consumption when running the test cases. In the Appointment class, I used the java SimpleDateFormat utility in order to simplify the need for comparing dates against the system date, which uses ParseDate to compare against todays date (new Date()) in line 21 of the class. This means that it is doing less checks and functions overall than if I had manually created a date formater and comparison function, but the class is now dependent on the java.util library.

**Reflection**

For testing techiques that were used, the most variation present in this project is in the Appointment and AppointmentService test classes. Mainly, for each of the classes, I use a parametered test case to check that a valid input can be recorded initially, and then follow up with each hard coded exception that was included in the main classes. Keeping the tests small ensures that any failures can be pinpointed easily, especially when following standard naming conventions for each of the tests. In the AppointmentTest class, I also utilized the @BeforeEach annotation, which allows me to simplify coding each of the test cases by eliminating redundant object creations and use only the tested function when updating the object with my setters. I could have done this with the other classes, but the appointment class was the most complicated with the valid date changing over time and having to manage what a valid input would look like in the test function. I should go back and use the @BeforeAll annotation at the very least on the remaining test classes in order to also simplify test case setup, and if we plan on expanding the test cases/application further, it would be a good idea to begin tagging the test cases with the @Tag annotation in order to manage readability when going through all the test cases.

In regards to my mindset during the development of this project and all its parts, I adopted a focus on meeting the requirements with a minimalist approach in all aspects. The less inputs needed to work, the less places that the code can fail. The simpler the functions and minimizing the code allows for the use of less resources, leading to more efficient code overall. With a minimalistic approach though, I also had to focus on not avoiding cautious practices, such as placing input verification exception arguements every time an input is received. It is redundant, and not 100% efficient, but necessary if we are to ensure the saftey of the system. For my own bias in how I developed the code, it is difficult removing your own thoughts and ideas of how things should work from the work itself, such as how originally my design did not include setter and getter functions for the main object classes, leading to less efficient and harder to read/test code. Having the feedback that was provided was extremely helpful, but software developers should hold a critical mindset with their own work. Writing the JUnit tests help with that bias in terms of failures can be tracked and fixed if defined well, but false positives can still slip in and be a cause for concern if you don’t open your perspective. Keeping that in mind, having discipline and not cutting corners is extremely important when developing and testing. This is mostly due to the fact that you never really know how your software will evolve and be used in the future, meaning that even small errors can lead to large, devestating impacts that you can’t possibly predict. In conclusion, developers have to balance caution and efficiency while avoiding limiting their perspective and foresight. If a developer manages their code well enough, it may interact or be used in other systems or applications further down the line, which raises the impact of errors or issues that may be present in the development process.