Summary and Reflections Report

While creating JUnit tests for the Contact, Task, and Appointment features, I believe my approach aligned well with the software requirements because I built the tests based directly on the requirements. An example of this is the Contact feature’s contactID attribute, in which the value of the attribute had to be unique, less than or equal to 10 characters, and not null. I built constructor tests for the Contact feature with the character limit and null value requirements in mind. I also made a contactIsNotUnique test within the ContactServiceTest class to ensure no duplication or overwriting occurs during the addition. For attributes that were editable, such as contactFirstName, I included a setContactFirstNameIsTooLong and setContactFirstNameIsNull test to ensure the attribute was not invalidated after construction.

By creating tests for all the software requirements, I was able to gain a coverage percentage of 100% for the grandstrand package, but only an 81% coverage percentage for the test package itself. I am still trying to improve my JUnit test coding capabilities to improve the overall coverage of 86% that I received. While this is still an acceptable coverage percentage, I would like to have code that has 90% or greater coverage for JUnit testing. The main area of concern when working with the JUnit tests where the percentage was hindered was when utilizing some of the “assertThrows” functions. When a test fails within the assertThrows function, the test passes but shows the function that would fail to be skipped in testing. If the test fails because the function within the test would pass, the test fails, and the coverage percentage goes down more. I will continue to work on this beyond this class to become a better tester.

To ensure that my code was technically sound, I did constant dynamic testing while fitting the code to the JUnit tests I built. Any logical portions of the code were tested as the classes were built, such as the following code:

if (Date == null || Date.before(new Date())) {

throw new IllegalArgumentException("Invalid Date"); }

This portion of the code ensures that the Date attribute is neither null nor in the past prior to the construction of the Appointment class object. I also ensured the code was efficient by creating functions where code would be used multiple times, which will prevent errors if the code needs to be modified later. An example of this is provided below:

if (!appointmentList.containsKey(appointment.getAppointmentID())) {

appointmentList.put(appointment.getAppointmentID(), appointment); }

This portion of the code utilizes the getAppointmentID() function created within the Appointment class to retrieve the appointmentID from an appointment object securely and efficiently, all while reducing the required future modifications to the code and improving readability.

While preparing the functionality for the Contact, Task, and Appointment features within the software, I implemented both static and dynamic testing. Static testing involves reading through the software and identifying any noticeable problems within the codebase. Dynamic testing involves taking a different approach and running the code to find issues. “The software application’s internal coding, design, and structure are examined in white box testing to verify data flow from input to output” (Ashtari 2022). Within dynamic testing, I utilized a white box testing method in which I understood the inner workings of the code prior to running the tests.

I did not implement any black box testing within this project. As I already understood the code I have written, I cannot test the code without knowing what is going on behind the scenes between the input and outputs. I also did not test the performance and security of the software. Since JUnit tests were utilized and no user interface was created, performance was of little concern. Security could have become a concern within the application, but I ensured that all attributes were private, and methods were created only for the required software features.

A practical use for static testing is that you can read through your or your coworker’s code and identify any security or logical issues that may not have been found while writing the code initially. For dynamic testing, you can run the code that you believe to be sound, but you may find out that there was a stack overflow error you did not account for when dealing with large integers. A practical example where the white box and black box testing can both be used is to ensure that if you provide a certain amount of money for shares in a company, you are provided with the proper number of shares, and no excess funds are transferred.

As a software tester, I took a much more cautious approach to coding than I have done in the past. Normally, I would have made sure the proper output was provided based on a given input. Now I can think more clearly to list out any possible scenarios in which an invalid output may be given and create tests for it. It was important to appreciate the complexity and relationships of my own code while testing it because a coder can be blinded by success and forget about the defects that may have been created. I did not initially understand how the Date object was utilized and had to work through creating the attribute and manipulating it before writing any tests for it. Once I understood it better, I could have written tests for its success, but not completed validity checks for null and character limit cases.

To limit bias in my code, I plan on utilizing the same process of gaining an understanding of new functions, then writing all the test cases I can think of in which invalid output might be created, and then writing the functional code based on those tests. Bias can be a major concern when writing your own code because it is always easier to write for the code to work instead of writing for the code to not break.

It is very important to be diligent and disciplined within one’s commitment to quality as a software professional because bugs and defects can negatively impact customers and our own lives. If a developer cuts corners, someone’s life savings could be lost, or a lethal dose of medication may be injected into someone’s veins. While most cases will not be this severe, there will be rework time required to fix bugs. Writing the code with the mindset of limiting coding faults in the beginning and testing for issues early and often is how I plan to avoid technical debt in my future endeavors as a software professional. I was very excited to learn about JUnit tests in this class and will be including test cases in a majority of my coding work to make sure I am not making errors as often as I get a job in the Computer Science field.

Reference:

Ashtari, H. (2022, September 29). *Black box testing vs. white box testing*. Spiceworks. <https://www.spiceworks.com/tech/devops/articles/black-box-vs-white-box-testing/>