**Project Two: Analysis and Summary of Grand Strand Systems Software Testing**

**Nathan Anglin**

In the previous project, I submitted code for Grand Strand Systems to meet the requirements for a task, contact and appointment scheduling system. Along with this code were Junit tests written to test and make sure all requirements are met and that the program’s architecture is running smoothly.

**Summary**

My approach for this project was to first address all requirements and begin to make a plan for how the program was structured. Using the requirements as a base, I began to think about how to make the contact service not only work, but to be encapsulated so that the code would be secure. In this case, I created accessors and mutators for all fields in the classes. This allowed the program to access information stored without any chance of changing values, and mutators to change values of variables without using global variables. After I decided on a class system with correct accessors and mutators for each field, I created a contact service class that would use these methods to change and update data. Once this was in place, I read through all the requirements of fields, such as “The contact object shall have a required firstName String field that cannot be longer than 10 characters. The firstName field shall not be null.” (Project One, 2022) Using the same coding practices, I did input validation for each field when any information was provided to the system. In the final product, these will be values that the user enters, and at that point there should be feedback from the program telling the user the rules for the field and ask them to re-enter information. For this fact I have all of the methods return Boolean values so error messages can be written later in the correct GUI used for the program.

For my Junit tests, I checked for every possibility I could think of. This included working and valid values being set, and one case for each invalid value set. I also tested right at the limit, for example for a string of length no longer than 10 characters, I tested for 9 and 11 characters in a string, as the boundary is usually where errors can be found, and this also tested base functionality. The coverage percentage for my tests was 98%, which means that 98% of the code had been run once all Junit tests have finished. While it is not always possible to get 100% coverage, the higher the coverage, the better. This shows that almost all lines of code were utilized, and the tests covered every failure condition, or points where the input validation catches an incompatible value for a variable.

Not only does this coverage provide technically sound code, it also insures that the code is efficient. With this 98% coverage we can see that there is no extra code in the program that is not being utilized. Also, as you can see on lines 18 – 48 in the ContactService class, I check for each condition that would flag the input as invalid, then immediately exit with an error so that no more code would be run. This saves unnecessary computation or saving of variables that would need to be erased. Along with this, all three groups use the same logic. This makes the code easier to change, as any changes that need to be made can be easily carried over to the other classes. As you can see Contact, Appointment and Task all share the same structure. The same is with the ContactService, AppointmentService and TaskService classes. For example, lines 50-62 in ContactService, 42-54 in TaskService and 43-55 in AppointmentService all share the same structure and work in the same way, searching for and deleting an instance of the object in the same way. This will help in later development if any changes need to be made to the classes. Between the three groups, the only major changes are the differing input validation and changes in how the variables are checked for validity.

**Reflection**

Throughout this project, I used various testing techniques, the first of which is unit testing, a type of white box testing, or testing while having access to the lines of code. This testing was done once a section was finished and allowed me to easily see if my code is working. This testing, which should be done often in the development process, allowed me to see mistakes in my code, both logical and typographical. After getting the full amount of test cases to run smoothly, and looking over the code I had written, I felt much more confident about my code’s ability to meet the requirements. After this the other main testing I did was integration testing, which happened when combining my three code bases into one program. This took quite a bit of work to get the folders working correctly, but once I had everything set up, I was then able to run my Junit tests again and make sure everything was still working correctly. In this phase I was able to make sure the correct files were linked between folders, and after crossing this hurdle, I was able to get my code running and passing all Junit tests.

In this project I did not do any black box testing. In this case, we do not have a working UI yet for the project, so black box testing is not possible currently. In the case that we had a working UI, I could run the program without knowing the contents of the code, and I could check for functionality and input validation. In this case it was not possible.

I think that with all these testing techniques, unit testing is in my opinion, always useful. In any coding project, you need to be able to test your work, and having Junit tests on hand to run quickly at all points during the process helps you catch bugs early. In larger projects, combining code from many different programmers fully utilizes integration testing, and making sure everyone is on the same page and the combined code runes smoothly. Finally, black box testing is a great way to test the final product and get feedback from users for changes to be made to the program. This can also narrow down any bugs that were missed by white box testing and may allow testers to look at the program from different angles.

My mindset for this project was developing the classes in small chunks, and testing these. I also used caution when working with variables and made a point to first read the requirements to decide how to set up accessors and mutators. I wanted to make sure that the code was not complex and that I did not repeat myself when coding. Keeping everything in its container made writing the code cleaner and easier to work with. For example, all Service type classes have access only to what they need to which is that related class. Within this relationship, the changing of data is closely monitored, and I made sure that every time data is changed, I again check for input validation.

It is a bit hard to eliminate bias when working on this project by myself, but I found coming back to the project and reading it over helped me look at it in a new light. I feel like once you have written a program, you naturally have a bias into the workings of the program and fresh eyes are good to see if there is a better way to do a task or spot any errors. For example, in my code I did not check both for empty strings and null strings, because after doing research the two are the same in Java. I could see that without this knowledge you could check for both, which would work, but would also add run time and complexity to the build. I personally waited to see my graded work, because although I was confident in my program’s operation, I could see a point I missed just because of my approach to the problem. I then reviewed by grading and adjusted along the way to make my code better.

I think being disciplined in your coding helps not only to create good habits, but also to create consistently better projects. One thing I have learned is that writing comments as you are writing the code helps you in the development process and is a good habit to get into. Sometimes it is tempting to write the code while it is on your mind and add commenting later, but I have found when I do this, my bugs start to become harder to find when I come back to the project. To reduce technical debt, I plan to be using unit testing on all my projects and stick to commenting at the time of writing the code. Between these two I won’t be pushing the buck down the road to a later date and keep my progress more consistent. Waiting to test and format correctly really doesn’t save any time, and the time you will spend in the future will always be greater when trying to remember the code and testing things you may have written weeks ago. For example, on the first part of this project I worked with the contact service. In this, I did not write my commenting down as I went, and when I came back to work on the project after a few days I had to completely read over my code again to write the comments. I wasted a lot of time just getting caught up to where I was, which is what the comments are there to do!

In the end I found this project fun and a great learning experience. I loved writing code from scratch and using that structure to write code for each assignment. Seeing the code come together and work well with good test coverage feels great, and I feel like constant unit testing is a great way to feel confident about your work. I look forward to more coding in the future and more challenges to tackle!

**Citations**

CS-320-H7221. (n.d.). Project One Guidelines and Rubric. Southern New Hampshire University. Retrieved August 11, 2022, from https://learn.snhu.edu/d2l/le/content/1116023/viewContent/19043220/View