# CS 320 Project Two Grand Strand Systems Project Reflection

The following is a report of the JUnit Testing approach for the recent mobile app project for contact, appointment, and task services. All tests are based on the requirements set by the stakeholders. For example, in LINE BLANK of the contact class, if the first name cannot be longer than a 10 characters, the test was written to test both first names that were shorter than 10 characters, equal to 10 characters and greater than 10 characters. In these test cases, the only option that should fail is the first name that has more than 10 characters. Using these methods to test both valid and non-valid options, we start to see how much coverage of the written code the test runs. This coverage number allows us to see how much of our code is functioning correctly based on our written tests. The coverage percentage of our testing was above 80% in the different areas which conveys that no errors were found within that 80%. Ideally, this number should be as high as possible and we should always strive for the highest amount possible, but there are bound to be areas that will be smaller in testing coverage in the first few iterations. Coverage is often checked with testing methods and in this case, we used JUnit testing.

As this was my first-time writing JUnit Testing programs, there was a bit of a learning curve on learning how assert and assertion commands work. The resources were very helpful in learning these while using examples online with how other programmers use these as well and making sure I understood the code and how I can use it myself. Ensuring these lines make logical sense, I had to think about how I was writing the code. For example, from the ContactServiceTest JUnit Tests, lines 25-31 focus on adding two contacts with unique IDs and then removing one of them. The assert command on line 31 is checking to see if only 1 item is in the current hash map. If the test passes, it shows that there is indeed only one item in the hash map and that it worked. If the test did not pass, generally there may have been an issue with the code somewhere when adding items to the hash map. During some of the initial iterations of the code during the milestones, the code was a bit inefficient but was changed during the completion of the project. In the first few iterations, the initial data structures to hold the internal data were arrays. Using arrays meant that loops would be used to iterate through the whole data structure and in some cases, depending on the size of the structure, this could take lots of time. In the TaskServices code, this meant using 8 lines of code to search through these. The recommendation was made to switch to hash maps as it may be more efficient by the supervisor, and it was changed. This reduced the number of lines needed from 8 to 5 and may have increased the efficiency of the look up times. These can be seen on the removeTask method on/around lines 22 of the TaskService code. To ensure the code was efficient and technically sound, a few testing techniques were used during this project.

There are many techniques available for different projects and only a select few were chosen for this project. Equivalence partition testing was used in this project to ensure that only valid entries that met specific number of characters were allowed to be accepted. This was one of the larger items looked at for each milestone. Requirements such as only allowing the appointDescription or taskDescription to be about 50 characters or less provided that they can only be somewhere between 1 and 50 and no more. There was also a requirement that it could not be null or empty otherwise it would not be accepted. Since the requirement was pretty exact with how many characters could be accepted, there was no opportunity to use bounty analysis here. Functional testing or some decision testing was also completed when attempting to add invalid items and seeing how the code would decide what to do based on these possible decisions from the input. In many cases, I’m wondering if this is where most of my coverage was lacking as there may have been decisions that were possibly missed in some of these decisions in the code. Some methods that were not used were methods such as the decision table. In this scenario, it was not required since there were not multiple decisions affecting the outcome of an item at least not altogether. As far as I understand, no structure testing was done either since there was no opportunity to build the portion of the of the user interface and how they will interact with the software other than just making sure the functional testing is there. Deciding which test to use will vary widely on the functions of the software. The more complex a program is, the more test that will possibly be needed in my opinion. Otherwise, if the application is straight forward or has very similar functionality throughout the application, the less test needed.

During the project, the mindset from being the developer of the code to being the test of the code is quite different. Under the development mindset, much of the time is spent getting the program to function as intended and once it begins to work, there is a sense of urgency to move to the next portion. Under the testing mindset, I found myself constantly questioning if the code was functioning correctly and if the testing was running correctly. I wanted to be very cautious and ensure that I was thinking of all the possible ways that I may get an error or false positives. Examples of this were during the testing of items that could not be null or had a limit in characters. Since some of logic of the code were using the same line of code to check both of these in one line, I had to ensure that during the testing phase, I wasn’t getting a the error from the wrong source. In this way, although my testing verification began to work, I still wanted to change up things and ensure I tested my test in different ways.

If the mindset shift from developer to tester does not happen, it can definitely cause a concern for bias in the testing process. Most of the time on the developer side, it’s easy to get caught up with once we get the program to function as it should to move on to the next item and to check the list off. If the same method is used when testing and just verifying that the test begins to work with only one type for that test, it can be easy to assume that the one item being tested means it works. This is the reason why during this testing process, multiple items that pass the test and multiple items that fail the test were used in the testing methods. For example, with the appointment service, to ensure that dates were only used for future dates for appointments we tested multiple instances with many different dates that would include dates prior to the current date and dates that were following the current date.

When it comes to writing and testing code, the quality of these process must be held to a high standard and held throughout the duration of the process. Just like the airlines hit multiple checklists before flights to ensure safety and quality, the same discipline must be taken in software development processes. Cutting corners or allowing lower quality software to be published or releases can lead to negative consequences for all parties involved depending on the nature of the software. There are plenty of examples in which lower quality software has led to data breaches, major accidents in which people may have died, money/revenue losses for business, and many more. Yes, these are extreme examples, but they do happen when sloppy code is allowed to be released. In some cases, there may be times that stakeholders may want a rushed product, but technical debt should be avoided at all cost unless to build a quick prototype that is understood to be a very initial look at an application/program. The planning phase should be the time to avoid any kind of technical debt. If items come up that were brought up by stakeholders last minute and need to be added last minute, these may be the times where technical debt might be okay since they may not have been originally thought of in the original planning. Technical debt may also be okay in the short term for the developer when iterating the first few lines and coming up with greater ideas or feedback when speaking with supervisors. I’m under the impression that as we become better as developers, our ability to avoid technical debt will improve over time with experiences.