While programming for Grand Strand Systems I wanted to strictly adhere to the requirements that they had laid out. To assist in this endeavor, I made sure that in each of the java files in the main package each requirement was written at the bottom. This allowed for easy reference and made for an impromptu check list. I repeated the same action in the test package to make sure that each requirement was tested and verified before submitting. I wrote several tests to cover each aspect of the requirements.

The resulting coverage was fair with an average of 60%. Testing was exclusive to the requirements and all requirements were met. However, this does bring some attention to why it’s not higher. There are certain elements that were added and not tested. For example, in ContactService.java, line 17 has a print Array List function that was not tested. This would need to be remedied by either testing the method or deleting it entirely. Other extraneous methods embedded in the code and not tested would also need to be remedied to obtain an 80%+ coverage score. With that being said, all requirements were tested and vetted multiple times and in different ways. In AppointmentTest.java, ContactTest.java, and TaskTest.java the creation of instances was tested to make sure not only that it was successful, but that it was done correctly.

To make the testing technically sound and efficient I employed a few different tactics. In the file ContatTest.java, in lines 12 – 43 I tested each individual aspect of the created contact. Separating them into their own tests allows me to identify which, if any, aspects are not functioning properly. Another way I made the tests efficient, can be found in TaskServiceTest.java. Here in line 22 you can see that the operation being tested is “task.getTask(“A001”).getTaskId()” the portion “getTask” was later incorporated to strengthen encapsulation and prevent issues by exposing the whole array list.

Of the various software testing techniques available to me, I used manual testing, JUnit testing, and functional testing. I used manual testing as I was coding to catch errors I was making. This varied form a missing semicolon to dereferencing a method when I needed it. Manual testing is among the most frequently used for me to prevent having to find errors later when it becomes more difficult.

The JUnit tests are what this class intended to teach me. By creating a test package with JUnit test files, I was able to test each file in the main package by testing specific lines of code and checking the outcome. This becomes handy when testing the system multiple times. It allows for repeated tests and checks for any exceptions that may be present. As the system continues to develop, additional tests are all that needs to be written, provided that the tests are written well.

I used functional testing to make sure that the system was complying correctly. Writing code for the main package, I assured that the system at least could mesh well with each other. This only tested to make sure that reference was done correctly and to make sure that everything was formatted as it should.

As for the tests I didn’t use, I didn’t employ usability tests due to the lack of user interface. Additionally, I didn’t perform an acceptance test for clients or end users, this would come into play at the end of a sprint or by a reporting time. Also, I didn’t use security testing which could be important as it prevents hackers and other unwanted users from accessing the system.

Hubris is the downfall for many. Believing you know everything and that your way is the only way is an easy trap for developers. Upon submitting my files the first time, I believed that, although not perfect, my tests had little to improve by. That changed with the professor’s critique of my tests. In ContactTest.java, AppointmentTest.java, and TaskTest.java I made the mistake of putting several assertions in a single test. I didn’t run into any problems, however if just one of the assertions was incorrect, I wouldn’t have had a clear answer as to why. So, to be cautious I made sure to keep tests down to one assertion. Tests need to be specific to be helpful, otherwise they would do nothing more than take up space.

Bias is a major concern when testing your own code. It’s easy to believe your code works exactly as it should 100% of the time. Conversely, it’s also easy to want to ignore something that was difficult to figure out with fear that if you test it, it will fail. To limit bias, I found myself trying to execute code that would work and code that wouldn’t. This would allow me to see if the standards were met in their entirety. The results were two to three tests for a single metric. It’s also reasonable to believe that in the future I will be responsible for testing my own code for a new system. Assuming that no other team member was available to help, I would plan on integrating two to three tests per metric. This would allow for multiple attacks on the system, and I would be able to tell when and how the system failed.

Being self-disciplined is easier said than done. There will be plenty of reasons to cut corners and many excuses as to why major mistakes were made. In my experience being self-disciplined is a continuous choice rather than a trait. When coding it is my responsibility to deliver the best product possible. My Father taught me that whenever I take a job my name will be attached to it. Whether my name matches a high-end product or a cheaply made one is up ot my discipline in doing the job. Aside from the marketability of myself, being self-disciplined will also make sure the system is working as well as it can be. This will also keep the system secure.