The process of software testing for the work done for our most recent customer I would say was a great success. The test I chose for the first module was a broad sweep unit test. This can be seen by how many more assertions there are 34, task and appointment both have 9. I realized this was not a good use of dev time as it was extra stuff the customer may not want to pay for, so I reduced the testing to only requirements for the next two modules. The accompanying service modules don’t follow this pattern as the ContactService was a very simple implementation and didn’t have any room to over-test. This is also reflected in the coverage and shows extreme diminishing returns for adding many assertions as the coverage of task and appointment are 90.9% and 89.6% respectively, the coverage of contact is 98.3% but that 8% boost came at the cost of around 4 times the assertions of the other 2 modules. To make this even worse the bonus tests didn’t cover any requirements and accomplished nothing in determining compliance. The only thing untested in Task and Appointment is the copy paste boiler plate that we already know works from the creation test, this accomplishes the same as Contact without wasting time with the boiler plate.

To ensure good technical code I first considered the responsibilities of the Service module and its base. The next step was considering how to make sure it had all the elements required by Grand Strand Systems, then I started writing. A good example of me following GSS guidelines can be seen in the AppointmentTest where I make a bunch of variables on lines 16-27 with simulated unique live data (Company requirement is to not use generic test data). I also make sure that the scope of the test is limited to the respective module tested.This can be seen in TaskServiceTest where the methods are directly named after each requirement. Speaking of requirements, the customer requirement document was an example of a compound requirement (multiple requirements in one), this is typically frowned upon and not good industry practice. Minor tangent aside, the tests were made efficient by only testing for each requirement while still retaining readability and modularity of tests so that there is a 1:1 ratio of requirement to test. AppointmentTest is a good example of this with a perfect 1:1 ratio (after breaking down compounds) for further review see lines 37,43,49,53,59,65,71.

The Software Testing techniques employed were unit/non-functional testing (Auto/Dynamic testing), functional testing(Auto/Dynamic Testing), code reviews (Manual/Static testing). Unit testing is looking and testing a specific module's whole functionality. In this case we used an automated form of this test in Junit. Functional testing is where you test to make sure a module is compliant with its requirements; this was done via Junit. Lastly, code reviews, I received feedback from QA telling me that they didn’t like certain aspects of my code and to change it. We didn’t do Integration testing or Regression testing. Integration testing is making sure a specific module can correctly communicate with other modules. Regression tests ensure that new code doesn't butterfly effect destroy some other module.

Unit Testing is often derided as being a lot of effort for little in return but it still has its uses. It is very good at highlighting edgecases that may have slipped your mind. It is also one of the easier ones for customers to understand and can help them feel more involved and secure in the development process. Functional testing is an absolute must on any commercial project as it is your way as a developer to prove that you’ve done your job. It will probably not find you any useful information about your project but it is one of those necessary evils to keep the project on track and organized. Code Review is going to be the place you have to learn some humility, I’ve been taken to school here too many times to not respect it. It is probably where I’ve been most enlightened to the weirdest and most esoteric feature of various languages and frameworks. It is also a place where you’ll be forced to deal with the arbitrary whims of whatever sadist reviewer that enjoys wasting time based on personal preferences. This one will always be useful despite the occasional pain. Integration testing necessity is directly correlated with a project's size and complexity. In a large and complicated environment these tests (usually automated) will help you spin up as a new dev and learn the code base without needing to bother the lead or someone more experienced. It is also useful to more experienced devs to give them a quick refresher on what the correct or the proper syntax for a less used API. Regression testing is pretty much always good practice and if you have time it is worthwhile to implement if your project has a long compile or start time. It will save code reviewer time or in the worst case if something made it past the introduction of new bugs into previously working modules.

This project reminded me of an old job I had where I used Cypress and Selenium. So I just brought back the ye olde strategy of reading the requirement doc and implementing tests. As a developer I was both making the code and the respective tests and I won’t pretend that I gave them equal weight in caution. I always tried my best to make my code easy to test and threw in sometimes disgusting constructors and or functions in prod code just to make the tedium of testing less bad. For this project it was much simpler so I was able to have my cake and eat as in I was able to not have a bunch of suspect functions in production and easily test my code. In the case of this code modules only acted in pairs with a clear hierarchy so in this case the code was very pattern friendly. The relationship of service and base had clearly defined roles with service acting as a controller (if this were a restAPI). This allowed tests to not need to have awkward overlaps.

My bias plays a huge role and testing being separate from me is probably going to lead to more discovery of bugs. When I code review with someone else they almost always find one thing that can be improved or fixed. This project is littered with me getting feedback for being slightly off the mark with what was intended, like forgetting the delete test case in TaskServiceTest. If I were better disciplined I would probably save a lot of mine and reviewers time. The temptation to speed through a project going based on what I think is the fastest route is immense and it ended up costing me time later on having to rewrite all of the Strings used in Contact for the final project. It took a good 30 minutes doing one of the most mind numbing tasks of changing all the Strings and then having to double check and test to make sure nothing broke. The best way to avoid this in the future in the real world is to make sure you’re keeping in contact with the team and always double checking you have access to the latest version of a document.