My junit tests are mostly written with an eye towards confirming the positive and negative test cases around each software requirement. If the tests run successfully with no errors I should as the developer know that my code meets the requirements set. For example, one of the requirements of our contact class is that it must have an id that is 10 characters or less and that the id string cannot be null. Based directly on this I created tests for instantiating new contacts that meet this criteria for either of the constructors in testBasicContact() starting on line 19 of ContactTest.java and in testFullyFormedContact() starting at line 32 of ContactTest.java. Those confirmed a new contact could be instantiated that met this criteria. I went on to confirm that instantiating a contact with too long an id OR with null as an idea would throw an assertion as part of testBasicContactIDTooLong(), testBasicContactIDNull(), testFullyFormedContactIDTooLong(), and testFullyFormedContatctIDNull() (lines 60, 68, 77, and 85 of ContactTest.java). I followed a similar approach to all other requirements, initially stubbing out the tests I would be writing based on the requirements that needed to be met and then writing the tests to verified both the positive and negative test case based on those requirements.

When checking the overall coverage for my project I'm seeing an average of 97.3%, well above the requested 80% for this project, which also happens to be the test coverage we should aim to be above according to our readings during the course: "As a general rule, a test coverage rate of 80% or above is considered reasonable" (Garcia, 2017, p. 284). My earlier testing passes were not this high as they did not necessarily test all potential paths that could occur at decision points, but I improved my testing code based on information returned about the coverage prior to the week 6 project. Based on the fact that it is now reaching 97.3% coverage and that the tests I've written are based on checking the positive and negative case for each requirement I feel fairly secure that functions I have written perform as intended based on the requirements I was given.

Planning my code out prior to implementing and reviewing after it was written for any errors or ways it could be improved is the primary way I ensure my code is technically sound. In the case of writing unit tests that planning includes making sure I am planning to write tests that cover both positive and negative test cases matching each of the projects requirements. Also keeping my code design to standardized object oriented programming practices helps keep it technically sound, such as employing encapsulation like can be seen in my Task class on lines 13 through 24 where the attributes for the class are set to private such that they are only accessible from inside the class, which can be done with associated setters and getters. During my code planning I try to keep an eye towards utilizing exiting design patterns, which is an area I am trying to improve. For example one opportunity that I missed during my initial planning was the that I would eventually be writing 3 different service classes that shared some mutual functionality. It may have been better practice to have planned out my code to implement a base "service class", possibly virtual as I would never intend for it to be instantiated, that contained the shared functionality and then have the Contact, Task, and Appointment service classes inherit from it.

Taking the time to analyze the requirements and design the code first rather than just diving in goes a long ways towards keeping my code efficient and reviewing the code and testing it after help here as well. In practice getting a second set of eyes on the design and on the final code prior to "merge", or in this case hand in, I find is helpful as well but I did not have access to that sort of peer review prior to turning in my assignments. Once again sticking to standardized object oriented practices and known design patterns when planning the code out is very helpful in creating efficient code. In this case as a specific example I made while planning the code I noted that the program was one that likely intended to store a large number of contacts, tasks, and appointments, and that searching for a specific one in the "list" would likely benefit from using key value pairing rather than a simple list array, which led me to use a HashMap tasks to store tasks on line 18 of the TaskService Class and similar HashMaps called contacts and appointments to store contacts over in the ContactService and appointments in AppointmentService.

I did however decide to verify that the ID generated for use with tasks, contacts, and appointments was unique prior to creating new tasks, contacts, or appointments and putting them into their appropriate HashMaps, generating another if it is not. This extra step does require an extra check which may not be necessary as a randomly generated 10 digit alphanumeric code is likely to be unique unless we generate a whole lot of them but I was paranoid about potential collisions and deemed the potential impact of such a collision worse than the additional time to check for this every time a new task, contact, or appointment is generated.

In order to address what software techniques were used or not used and the details about them I'll start by looking at the division of static versus dynamic testing techniques. Static testing techniques of course "are those techniques that test software and work-products without executing the code" (Thompson, 2015, p. 62).

Static testing techniques include reviews, of varying formality, which usually require someone other than the initial developer to be involved to get the full benefit of them. It is arguable however that I have performed one or more informal reviews of the code I've submitted as I visually checked them for bugs and inconsistencies in design or practice. I certainly found and corrected a number of errors in this manner prior to ever running my code to test it, however a lot more benefit is gained by getting a different set of eyes to perform at least an informal peer review before code submissions which is something I lack during these assignments. Another form of static testing technique is analysis by tools. I have not been performing this with the current code but have utilized software tools in the past to find potential security issues with imported libraries and dependencies on other projects which can be very valuable and isn't necessarily something that would be easily found during dynamic testing or visual code reviews.

The Junit tests we've been creating on the other hand are dynamic tests and require us to execute the code in order to run the tests. These test techniques come in three different categories:

'Black box' techniques, also known as specification based techniques which I prefer as a name, 'derive test cases directly from the specifications or from some other kind of model of what the system should do' (Hambling, 2015, p. 85). I have made use of this a lot in the Junit tests that I have written. For example, I have been given the requirement or 'specification' that new appointments must have a unique id and that unique id must not have more than 10 characters. To test this specification, the testBasicAppointment function on line 20 of my AppointmentTest class instantiates a new appointment with an acceptable id length and then gets the appointment ID of that appointment and confirms it equals the appointment ID we set it to. Another function on line 61 of the same class, testAppointmentIDTooLong runs the negative test case and confirms that when one attempts to instantiate an appointment with an 11 character id length an assertion is instead thrown. Both these tests are specification-based. A well designed system should meet all the specifications that were set for it in design making specification-based testing very important, but its important to keep in mind that this testing verifies that specifications are met and not how specifications are met. As our text puts it 'The most important point about specification-based techniques is that specifications or models do not (and should not) define how a system should achieve the specified behavior when it is built.' (Hambling, 2015, p. 85). Five specification-based techniques are equivalence partitioning, boundary value analysis, decision table testing, state transition testing, and use case testing.

'White box' techniques, also known as structure based techniques are 'used to explore system or component structure at several levels'. (Hambling, 2015, p. 100). These include the individual component level, the integration level and the full system level. Because I am looking at what percentage of code coverage I am getting with my junit testing and confirming that all branches of decision events are being hit as part of that coverage to a certain extent my tests are also performing some light structure based testing (all the statements in my component are run as part of my tests without errors occurring due to them. It has not been a strong focus of my testing however.

The final category of testing techniques is 'experience based techniques' which are 'based on deriving test cases from the tester's experience of similar systems and general experience of testing' (Hambling, 2015, p. 84). Two forms of this are error guessing, where testing is done based on experience with similar programs to find errors that might otherwise be overlooked and exploratory testing. Neither of these are techniques that I have utilized in testing during this class although both are techniques I have utilized during my time in QA and in my experience are more likely to be the providence of QA testers than developers. Even then the value of error guessing really depends highly on the experience of the tester and isn't the sort of testing I would rely I heavily. Its a nice bonus if you happen to have time and the tester involved has a hunch they would like to explore to find a potential problem in my opinion.

I find both writing code and testing benefit from a rigorous approach and mindset, which I suppose is similar to a cautious approach. I believe testing should be planned rather than ad-hoc whenever possible and should match up to the requirements of the program or code fragments being tested. Both positive and negative test cases for all requirements need to be covered and the tester needs to show an awareness things like decision branching, looping, inheritance, really everything the same knowledge set that was necessary in initially writing the code, in order to be able to write tests that are comprehensive while remaining efficient and avoiding extraneous writing extraneous tests that don't increase confidence in the code running as intended. For example when writing tests for the contact service to verify its requirements for a id key length it is important to recognize that the if statement decision point on line 43 of ContactService.java will require 3 test cases to make sure all possible branches are confirmed working and cover both the positive and negative test cases. Similarly it was important to be aware that when writing negative test cases to confirm that we cannot create a contact with less than 10 characters that we only need to confirm that it does not work for 9 characters. We do not also need tests for 8, 7, 6, etc characters. If the test passes, by throwing an assert when we try to instantiate a contact with 9 characters, we know it would also throw an assert for fewer characters.