My unit testing approach for completing the three features of the software project was to consider having a test that exercised each requirement. Doing so adds rigor to your tests and gives the development team, product team, and the testing team confidence that the software does what it is supposed to do. For example, specific requirements made it necessary that certain attributes fit within length constraints. Therefore, my unit tests were specifically designed to catch instances where the correct exception was not thrown when it was supposed to in order to ensure that users could not enter strings longer than was necessary. There were also specific requirements to be able to add an appointment/contact etc. given a unique id. I specifically created tests to ensure that classes could be properly instantiated given an ID.

My Junit tests were comprehensive in that they covered all of the necessary requirements adequately such that, if one of the tests failed, developers would know a regression had been introduced to the code which renders the software program incapable of executing on its essential functions. Most of my classes had 80 percent or higher line test coverage which is often seen as the minimum standard for coverage. In cases where coverage was less than 80 percent, those files were scrutinized until a reasonable explanation could be discerned as to why a threshold of lower than 80 percent was ok. In my instance, tests were not created for getters and setters. It was deemed ok not to create tests for these getters and setters as they did not perform any business logic, so testing is not necessary for them.

My Junit tests were technically sound and efficient because they took advantage of built in test-construction design patterns such as setup and teardown methods as seen at the top of each one of my test classes (AppointmentServiceTest line 16). This ensured that my tests were concise, reusable, and readable. This contributed to the overall quality of the tests because in a software project it is critical to apply the same software engineering principles to the tests as one does to the code.

During this software project, I utilized unit testing, boundary analysis, and input partitioning. Unit testing is the process of testing each piece of your codebase on a method level and should not include things such as connecting to other methods, connecting to a database etc. It’s meant to test functionality at a low level to ensure the code is working from that standpoint. Input partitioning is a methodology to help the tester brainstorm specific values to use as input in their tests. Correctly chosen values can reduce the number of tests that need to be written as conducting a test on a value of *x partition* tests all values of that partition. Boundary analysis, which employs input partitioning, is the method of selecting values that are right on the edge of being acceptable for the code under test. One specific example of this from the project is the requirement that “Id cannot be longer than 10 digits”. Instead of choosing a random string of digits longer than 10 digits, a string of 11 digits was chosen as it was just outside the range of what is acceptable. Full utilization of this technique dictates that a number just within the range of acceptable must also be chosen. Although full utilization was not realized, the technique was channeled to find higher value inputs. Unit testing can be used by developers at all stages of the SLDC to ensure that regressions have not been introduced into their code given new code pushes, and input partitioning and boundary analysis can be used within those unit tests to increase their rigor.

Other software techniques that were not used in this project were decision table testing and state transition testing. Decision table testing is used to design test cases and involves creating a table where conditions are listed at the top and actions as a result of those conditions are written at the bottom. It allows a tester to prove that they have tested all of the conditions that a computer program can be in or all the relevant conditions that it can be in.  Another technique closely related to decision table testing is the creation of a state table. A state table is a diagram that shows all events that can occur inside of a system and the result of those events. It focuses on deriving system behavior from the transition between states. This can also be used to generate test cases. For example, one can diagram a watch’s functionality by showing what event (pushing a button) causes what state. Both state diagrams and decision table testing can be used on all software projects as a general technique to derive test cases in a meaningful, measurable, systematic way.

I employed caution throughout the course of this project by assuming that every piece of functionality that executes some part of the business logic can and will fail. I thought to myself what are the ways in which a user can enter data that I would not expect? How does my software react to it? For example, specifically regarding my code, a user could enter an id that does not exist when trying to delete an Appointment or Contact. It would be catastrophic if in the event a customer does that, my program deletes the first contact in my program. Therefore, I ensured (not using a unit test but just doing exploratory testing) that this would not occur.

I tried to limit bias when reviewing my code by trying to put myself in the shoes of a tester. By assuming defects are present and writing tests to coerce those defects I can help eliminate the temptation of just writing happy path tests that may not have the necessary rigor to catch the bugs I need them to catch. Bias could definitely play a role if you need to test your own code. Your own code often has a personal connection with you. Which makes it difficult to see the flaws in the code. Also, time pressure and a natural propensity to want to build new things can raise the risk of overlooking testing the current code. For example, on a personal project I felt immense pressure to move on to a new requirement that I was interested in rather than writing unit tests for existing code because seeing my project take shape gave me much more motivation to keep going than writing unit tests did. However, in the long run, implementing new features with no unit tests will cost me more time in the long run.

It is important to be disciplined in your coding and to not cut corners as a software developer because there is always a high chance that the effects of that will be seen down the line. A lack of tests here or there may cause a defect to be introduced later in the code that could have been rectified much earlier in the project with some testing. And, the later bugs are identified in the project, the costlier they are to fix. This causes much waste on the company’s part in terms of money and resources. I plan to avoid this technical debt by writing unit tests early and by carrying out other forms of testing such as static testing to make sure any defects are prevented or caught as early as possible in the SLDC. A specific example of how this can be done is by identifying an assumption in the software requirements that causes the engineering team or product managers to rethink the requirement before any code is written. This saves the developers headaches later and saves the company money in the long run.