I believe that my testing approach was thoroughly aligned with the software requirements. The software requirements were plainly explained in the assignment instructions which gave concise and clear instructions on what was to be expected from the software. This allowed the testing process to follow the instructions line-by-line when designing the tests to ensure that the testing process provided full coverage of each of the given requirements. I was able to create the JUnit tests for some of the requirements given and others required manual code review for testing since they were unable to be tested via a JUnit test case. An example of a requirement that was unable to have a JUnit test case is the requirements that defined the program’s Task class to have required fields. This was executed by defining the constructor for the class to only be created with those fields. Therefore, an instance of the class would not exist without those fields. Furthermore, the JUnit test cases that were applicable to this requirement tested if these fields were null; thus, the fields were both defined during instantiation and not null. One requirement that eluded me when designing this program was the one that defined the task/contact ID to be unique within its respective class. This was also a requirement for adding an instance (task/contact) to its respective service instance (taskservice/contactservice). I enforced the requirement when adding the instance to the service instance instead of when instantiating the base class itself. This results in the arraylist used to track the instances to never have a duplicate ID, but it is still possible for the program to create multiple instances with the same ID. In order to check for duplicates in the initial class construction would have required either more complex code design or an additional class. I assumed for the scope of this assignment that the software requirements were still met even with this slight hole in the logic.

I believe that the JUnit tests that were created for this program did a good job of assuring the quality of the software was acceptable in terms of the given requirements. All of the test cases provide near 100% coverage of the required classes they were testing. The only issues that I noticed when checking the coverage percentage is that I did not create tests specifically for some of the getter and/or setter methods for the class variables in the Task and Contact classes. However, the logic that I used in these methods for authorizing the input for setting the member variables was the same logic that I had tested for in the constructors for the classes.

In order to ensure that the code is technically sound, I followed all of the requirements given in the assignment, I compiled and ran the program multiple times, I ran the test cases multiple times and check the coverage, and I followed basic Object-Oriented Programming Principles in order to build a solid program. Some of the features that allowed for this basic structure was creating member variables and methods and constructors for these classes. The constructors for the base classes involved most of the business logic for the program which allowed for the rest of the program to be easily built around it. The getters and setters for these classes were simple and built off the logic already developed in the constructors. One thing that was absent from this program that was not a requirement that I believe would have made the program more technically sound and useful moving forward would be accessors for the ArrayList object that is used to store all of the base class instantiations. The mutators allow for editing the information, but the lack of accessors prevent the data from being used in any way.

When it comes to the efficiency of the program, I believe that my code was decently efficient. I could’ve made the program more efficient, and in turn provided full coverage during testing, by using the getters and setters in the constructors for the class. This would have prevented the re-use of the same business logic. Given the scope and simplicity of the project I didn’t think it would clutter the program too terribly by not doing this. Otherwise I believe that my test cases were efficient. The only question that I may have about their efficiency is my use of the for-each loops when checking the ArrayList. I believe these to be decently efficient for iteration, but there may be a more efficient data structure/iterator combination that would provide better performance.

I would say that the main software testing techniques that I utilized during the milestones thus far would be black-box testing techniques. Black-box testing involves testing based on the given software requirements of the project. Since the milestones thus far have involved designing the programs specifically based on the software requirements, black-box testing techniques have been the ideal choice. Of the different black-box techniques identified in our textbook, I believe that equivalence partitioning and boundary value analysis were the two main ones that I utilized. Equivalence partitioning groups data values into separate categories or containers. This type of testing was executed commonly during the testing of the milestone submission so far due to the heavy input-related requirements such as, “the string cannot be longer than 10 characters long”, or, “the appointment date cannot be in the past”. These requirements allow the data to be split into separate categories such as strings between 0 and 10 characters long and strings longer than 10 characters or dates before current and dates after current. The other common testing method, boundary value analysis, was also incorporated with these same testing methods. Boundary value analysis involves analyzing the data near the boundaries of these given data categories to check for errors. These errors are common when dealing with relational operators and loops. For example, if checking if a string was not greater than 10 characters this testing ensures that the programmers didn’t accidentally choose the wrong relational operators such as < or >=.

The other two techniques that are identified in our text that I did not use in the milestones thus far are white-box testing and experience-based testing. White-box testing involves testing the program for complete coverage. This ensures that every line of code: all statements, conditionals, and loops are covered in the testing process. There is an argument to be made that white-box testing was utilized in my code for the milestones since they covered mostly every line of code, but the reason that I did not include this is due to the fact that it was more a side-effect from the black-box testing rather than a sought-out testing technique. Experience-based testing involves building tests based on previous knowledge of common errors that programmers or code may make/have.

The three different types of testing techniques, black-box, white-box, and experience-based all help to achieve different types of testing success. Black-box tests are designed to ensure that the program is accomplishing what it is intended to do based on the software requirements. These tests are the backbone of any program testing. White-box tests are designed to ensure that every line of code is covered during the testing process. These tests add additional support to ensure that the program operates as designed during every conceivable use-case and state. Experience-based tests are the final step in the test-building process and are what help to separate the good tests from the great tests. These tests help to find the smaller, more specific issues with the code that wouldn’t generally be identified by normal methods.

I would say that the mindset that I had going into this project was a creative and inquisitive one. I believe that I did employ caution during this project as well. It is a healthy balance to be creative and cautious while developing a program as well as testing it. The way in which the testing results were analyzed exemplifies how I employed caution during this project. Utilizing black-box testing techniques ensured that the program produced the desired results; the white-box testing results i.e. test coverage, ensured that all parts of the program were checked. I believe that the creative aspect of my mindset was used when assessing the complexity and interrelationships of the program. It requires time and thought to view the program from different angles and how it interacts with different parts of the program. One part of the program that required some creativity and thought was testing how the program utilized the Java Date class. This class is a little outdated and has been replaced with the Calendar class in many instances, so it was fun and interesting with how I should interact and test the use of this and testing separate Date instances. Another example of how the complexity of the code required extra cautiousness and creativity was displayed in the AppointmentService class and when to throw exceptions. In the addAppointment function, the loop is used to check if an exception should be thrown, but in the deleteAppointment function, the exception is thrown if the loop fully iterates and exits.

I tried to limit bias in my program by executing the testing process as if I didn’t develop the code and look for ways in which the program could be faulty. I believe that the test coverage results display this view. As a developer, you believe that every aspect of the code will produce the expected results, even the small parts. As a tester, you have to test all parts of the code, even the small parts to ensure that it all acts as expected. This is where bias can impact the testing process if it is executed by the same developer. In my program, I even fell victim to this bias when developing the first parts of the program. When I developed the Contact class, I used the constructor of that class to assign the member variables and then I also had separate accessors and mutators. During the testing process, I only tested the constructor to ensure that it properly assigned those member variables with the correct values and not the mutators. I did this because as the developer I used the same lines of code in the mutators that I did in the constructor, therefore, I believed I did not need to test both sections. This bias can lead to issues due to potential errors with how the code could interact with these different functions. In the later sections of this project, I decided to utilize these mutators in the constructor like I did for the Appointment class. This way, when I tested the constructor for proper assignment of member variables, I was also able to test the mutators at the same time because it directly called these functions from within.

I believe the importance of being disciplined in the commitment to quality as a software engineering professional lies in wanting to be a good coworker and employee. This results in the avoidance of technical debt and its repercussions. Technical debt refers to the amount of work that is required to rework code that is faulty. This technical debt can be accrued due to not being cautious and often working too fast for what the project requires. The consequences of this in relation to being a good coworker is understanding that your team is responsible for the program as a whole, and any issues that the program has will result in your team having to spend more of their time going back to fix. With the relation to being a good employee, any time that your team has to spend on going back to fix issues, will cost your employer more money and time on the project which hurts their bottom line and reputation with the client. In order to avoid this technical debt in a real world setting as a software engineering professional, I believe that it is important to collaborate effectively, be cautious in your development, and spend the appropriate amount of time on each step in the software development life cycle. By collaborating effectively, you can ensure that multiple people can check and approve specifics of the project to reduce the amount of mistakes that a single person may overlook. Caution while developing can produce the same results of reducing mistakes, and it can require a developer to require less team involvement and improve efficiency in the long run. These both can be incorporated into giving each step in the SDLC its appropriate time and consideration. Since the designs that are created at the beginning of the process will yield how successful the rest of the project is, it is important to ensure that an appropriate amount of time is spent during this process to ensure the rest of the project runs smoothly.