**Summary**

The software for the Contact, Task, and Appointment objects and services was designed and developed around meeting each of the requirements provided. To ensure that I covered each requirement properly, I first made sure I had a thorough understanding of what the deliverables were. For example, in the Appointment object, the requirements specified that the appointment id property is required, must be unique, and cannot exceed 10 characters. The appointment id is also not allowed to be null and cannot be updated. The appointment object also needed to have a date field and the property is not allowed to be in the past or be null. Lastly, the appointment object needed to have a required description property that could not exceed 50 characters, nor could it be null.

Before any development began on the appointment object, I first read and analyzed the requirements in its entirety. Once I had a clear understanding, I began the initial round of development to create the appointment object. Given that two of the three properties here were strings and had relatively similar requirements at a conceptual level I figured I would be able to create a method to handle validation across properties. Since the appointment id and description properties share most of the same requirements, I began writing the validation methods accordingly. First, I was able to combine the requirements of properties being required and not being null into a single method named *IsNullOrEmpty()*. This ensures that the string property cannot be set to a value of null, as well as the property needing to have some value provided outside of an empty string. Next, I created another method named *IsStringValidLength()* which accepted the string to be validated upon, as well as a desired character limit as a maximum length. Using the *.length()* function from the String class, I was able to get the number of characters from the string passed in. Using the greater than comparison operator, I validated that the number of characters did not exceed the intended maximum length. The oddball here in this object included the appointment date field as this property was not a string, but a date so it needed a separate form of validation. I could have updated the *IsNullOrEmpty()* method to accept an object instead of a String, but I felt it was best to use strong types when applicable. Because of this, I created a new *IsDateNull()* method which ensured the date property had a value populated. Lastly, I created a method named *IsDateInPast()* which made use of the Date class’s *.before()* function which took in an instantiated Date object to ensure the date passed into this validation method did not have a value that was previous to the instantiated date object.

The level of test coverage I had on the concrete classes covered all of the requirements. I could actually argue that each class implemented with the application was completely covered, although I believe the Appointment Service had something similar to 99.7% coverage. This was due to a situation with the *.RemoveIf()* function on the array list used in the class allowing a scenario in which if the parameter sent into the function was null it would throw an exception. My argument for it being covered is that a check for a null or empty property happens before the .*RemoveIf()* function takes place, so the exception would likely never get hit. From a coverage standpoint, I still believe that the coverage level it sits at now is still very high and well above a safe limit to deploy the code into a production environment.

As I mentioned earlier in my summary, after getting a thorough understanding of the requirements, I began my initial round of development. When going through any product implementation, it can be expected that the first round of code written will not be the final production code. Code should be written first to meet the requirements, and then revisited afterwards to look at opportunities for refactoring or enhancing code to be more performant. An example of having technically sound and efficient code can be seen in my implementation of validation methods. It would have been easy enough and most likely would have produced the same results to have all the validation included in the object constructor, but in order to reduce duplicate code and follow a coding principle known as DRY (Don’t repeat yourself) I found it to be a good opportunity to break them into their own validation methods and call them accordingly. Taking it one step further could have included creating an extension methods class and moving the methods re-used within each object into this new extension class and had a single location for each object to call for. There’s always room to improve or refactor code.

**Reflection**

Throughout the development of the Contact, Task and Appointment services, and respective objects I took a dynamic testing approach. Dynamic testing can include, but not be limited to, creating unit tests and verifying results upon code compilation. I did this dynamic testing in an iterative approach as mentioned above in my summary section in that I developed small chunks, then covered the new code appropriately with unit tests and moved on once the code was working as intended.

While I did manage to make usage of the dynamic testing approach, I did not take any static testing approaches. Static testing is usually revolved around peer reviews or reading over the code from a syntax perspective to find any potential errors and address them before the code is compiled and ran. Each style of testing has their own benefits and purposes though. I personally enjoy a dynamic testing approach as sometimes unforeseen consequences can occur during compilation which may not be entirely obvious from a visual perspective. With that being said, static testing such as peer reviews are essential. Other opinions and thoughts not only help ensure the code meets requirements but offer great learning opportunities to increase one’s knowledge base from other developers.

When writing any sort of application or developing a product, it’s vital to remain as un-objective as one can about your code. No code is perfect, and I can guarantee that if I didn’t take the time to review all requirements both before and after I surely would have missed items that needed to be covered. I think developers often go into situations like peer reviews and think their code is full proof because they hit a certain level of coverage. In almost all situations code can be improved upon or alternate routes can be considered to reach the same outcome. In the situation here, I am the lone developer working on this application so proper steps needed to be taken in order to make sure that I limited my mistakes to be as few and far between as possible. After writing all the implementation code and covered properly with unit tests, I re-read all the requirements to double check my work to ensure everything was as it should. I plan to implement practices like double checking my work after it’s written but before it’s deployed into my career as I feel it’s good practice to give yourself the realistic vision that you may have missed something on first run through. Deploying something into production that may have missed a requirement will take a lot more time to resolve than double checking, or even triple checking your work.